NetworkxDemo

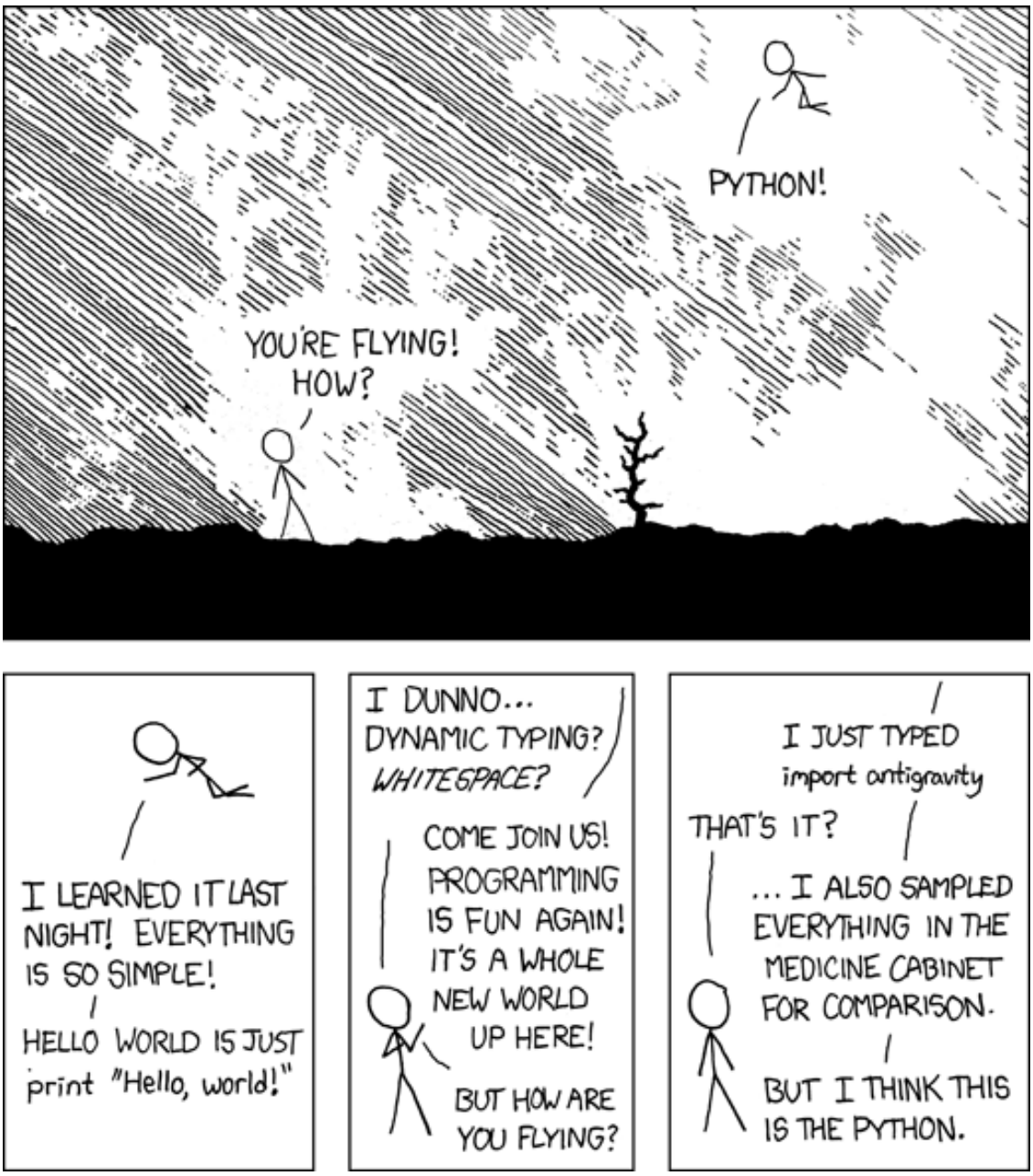
April 28, 2014

1 An introduction to network tools in Python

Python has been a very popular choice for a scientific programmming. I could say many things about it but I feel the following XKCD comic pretty much sums it up.

```
In [122]: from IPython.display import Image
          Image('http://imgs.xkcd.com/comics/python.png')
```

Out[122]:



1.1 Useful modules (or libraries)

Numpy and Scipy are useful for standard scientific computing. Networkx and iGraph are more specialized for networks. Matplotlib is a great plotting library. For this introduction we will not be requiring functions from Scipy or iGraph. The links to all these modules are can be found in the list of resources.

```
In [123]: import numpy as np
          import networkx as nx
          import matplotlib.pylab as plt
          from __future__ import division
```

1.2 Basic graph operations in Networkx

```
In [124]: g = nx.erdos_renyi_graph(10, 0.5) #create an ER random graph with 10 nodes
          # and probability of connection = 0.5

In [125]: g.add_node(12) #add a node

In [126]: g.add_edge(12,"A") #add a new node "A" and connect it to node 12

In [127]: nx.connected_components(g) #list the connnected components of the graph

Out[127]: [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], ['A', 12]]

In [128]: g.degree() #return a dictionary of nodes and degree

Out[128]: {0: 3, 1: 5, 2: 5, 3: 4, 4: 4, 5: 4, 6: 3, 7: 8, 8: 4, 9: 4, 12: 1, 'A': 1}

In [129]: nx.clustering(g) #list the clustering coefficient for each node

Out[129]: {0: 0.6666666666666666,
           1: 0.4,
           2: 0.4,
           3: 0.16666666666666666,
           4: 0.5,
           5: 0.3333333333333333,
           6: 0.6666666666666666,
           7: 0.35714285714285715,
           8: 0.5,
           9: 0.3333333333333333,
           12: 0.0,
           'A': 0.0}

In [130]: nx.betweenness_centrality(g) #compute betweeness-centrality for each node

Out[130]: {0: 0.00606060606060606,
           1: 0.04696969696969696,
           2: 0.04090909090909091,
           3: 0.034848484848484844,
           4: 0.016666666666666666,
           5: 0.03787878787878787,
           6: 0.00909090909090909,
           7: 0.1712121212121212,
           8: 0.028787878787878786,
           9: 0.025757575757575753,
           12: 0.0,
           'A': 0.0}
```

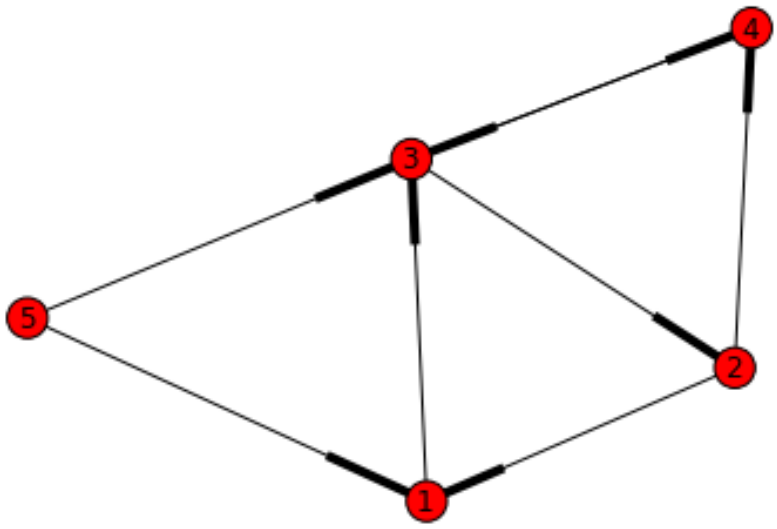
1.3 Random walk on a graph

We will compute the steady state distribution for a random walk on a given network. We will also learn how to visualize a simple network as part of this exercise.

```
In [131]: g = nx.DiGraph() #define g to be a directed graph

In [132]: g.add_edges_from([(1,3),(2,1),(2,4),(3,2),(3,4),(4,3),(5,1),(5,3)])
          #this is the same graph used in HW1, problem 2

In [133]: nx.draw(g) #visualize the graph. The thicker stubs are used in place of arrows.
```




```
In [134]: M = nx.adjacency_matrix(g) #obtain the adj. matrix for the graph
```

```
In [135]: print(M)
```

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

```
In [136]: for i in range(5):
              if (np.sum(M[i]) > 0):
                  M[i] = M[i]/np.sum(M[i])
```

```
In [137]: print M #normalized matrix
```

```
[[ 0.   0.   1.   0.   0. ]
 [ 0.5  0.   0.   0.5  0. ]
 [ 0.   0.5  0.   0.5  0. ]
 [ 0.   0.   1.   0.   0. ]
 [ 0.5  0.   0.5  0.   0. ]]
```

1.3.1 Numerical approach

If we denote the probability of finding a random walker on each node as a column vector, the transition matrix T is given by:

```
In [138]: T = np.matrix(M).transpose()
```

In order to find the stationary distribution start with a state vector with equal probability on being on any of the nodes and repeatedly apply the transition matrix. In the limit of number of times going to infinity the resulting state vector gives us the stationary distribution. For small graphs, the number steps required is quite small.

```
In [139]: state_vec = np.matrix((1/5)*np.ones(5)).transpose()
          #start with equal probability of finding a random walker on any node

In [140]: for i in range(15): # here 15 is the number of times we apply T on the state vector.
              #choosing a small number to illustrate a point.
              state_vec = T*state_vec
          print state_vec
```

```
[[ 0.09960938]
 [ 0.2       ]
 [ 0.40078125]
 [ 0.29960938]
 [ 0.        ]]
```

```
In [141]: #check if state_vec has converged
          print T*state_vec
```

```
[[ 0.1       ]
 [ 0.20039062]
 [ 0.39921875]
 [ 0.30039063]
 [ 0.        ]]
```

```
In [142]: # since the vector obtained above is not the same as the state_vec,
          # apply T on state_vec a few more times
          for i in range(35):
              state_vec = T*state_vec
          print state_vec
```

```
[[ 0.1]
 [ 0.2]
 [ 0.4]
 [ 0.3]
 [ 0. ]]
```

```
In [143]: # check if state_vec has converged.
          print T*state_vec
```

```
[[ 0.1]
 [ 0.2]
 [ 0.4]
 [ 0.3]
 [ 0. ]]
```

Since the vector has now converged this is the stationary distribution. Note that the vector obtained after 15 iterations was still a good approximation due to the small number of nodes present in this network.

1.3.2 Analytical approach

We know that the steady state distribution is related to the spectrum of the normalized adjacency matrix. In this part we illustrate this approach.

```
In [144]: import numpy.linalg as la #import linear algebra module

In [145]: T = np.matrix(M.transpose())
          print T

[[ 0.   0.5  0.   0.   0.5]
 [ 0.   0.   0.5  0.   0. ]
 [ 1.   0.   0.   1.   0.5]
 [ 0.   0.5  0.5  0.   0. ]
 [ 0.   0.   0.   0.   0. ]]

In [146]: eigvals, eigvec = la.eig(T)

In [147]: print eigvals
          #list the eigenvalues. The eigenvector corresponding to the eigenvalue of 1 is the one that g

[  1.00000000e+00+0.j   -5.00000000e-01+0.5j  -5.00000000e-01-0.5j
   2.28576394e-17+0.j    0.00000000e+00+0.j ]

In [148]: print eigvec[:,0] #eigenvector corresponding to eigenvalue of 1

[[-0.18257419+0.j]
 [-0.36514837+0.j]
 [-0.73029674+0.j]
 [-0.54772256+0.j]
 [ 0.00000000+0.j]]

In [149]: steady_state = np.real(eigvec[:,0]/np.sum(eigvec[:,0]))
          #normalize to get steady state. Note that the imaginary part is zero, which it should be if e
          print steady_state

[[ 0.1]
 [ 0.2]
 [ 0.4]
 [ 0.3]
 [-0. ]]
```

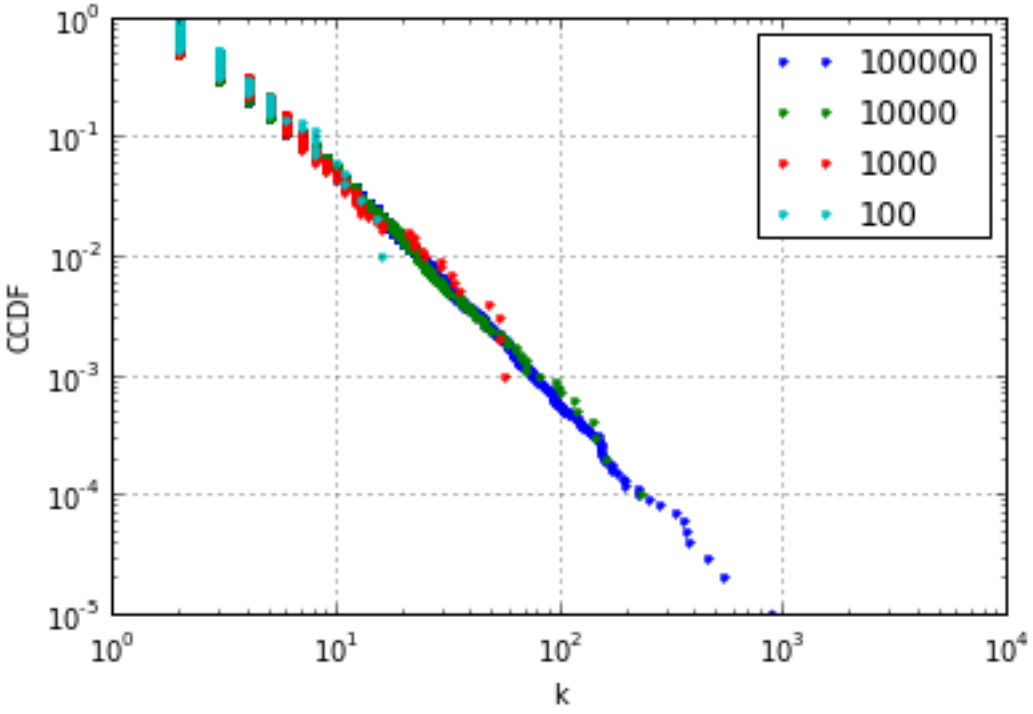
1.4 Finite size Effects

In this section we explore the effect of finite size in networks by studying the degree distribution of the Barabasi-Albert graph.

```
In [150]: for N in [100000, 10000, 1000, 100]:
              g = nx.barabasi_albert_graph(N, 2) #generate a Barabasi-Albert graph with N nodes with ea
              k = np.sort(g.degree().values()) #get the degree of all nodes in the graph and sort them
              y = 1 - np.arange(1., N+1)/N # probability in terms of a rank order
              plt.loglog(k,y,'.',label='N') # plot the CCDF. This makes the tail easy to visualize

          plt.xlabel('k')
          plt.ylabel('CCDF')
          plt.grid() #switch on grid
          plt.legend(loc=0) #show the legend and use the best location for it

Out[150]: <matplotlib.legend.Legend at 0x100c9090>
```



The figure above shows how the degree distribution appears to be more heavy tailed as we increase the system size. Recall that the largest degree that can be present in any network is limited by the total number of nodes present. It is important to test and account for finite size effects when we study a system based on simulations.

1.5 Links to the software packages mentioned

Here are the links to all the modules mentioned in this notebook. The links below will take you to homepage of the modules. It should be easy to find tutorials and installation instructions there.

Python: https://www.python.org/
IPython: https://www.python.org/
Numpy: http://www.numpy.org/
Scipy: http://www.scipy.org/
Networkx: http://networkx.github.io/
Matplotlib: http://matplotlib.org/
iGraph: http://igraph.org/

In [150]: