LA-UR- 08-5495

Approved for public release;
distribution is unlimited.

Title:	EXPLORING NETWORK STRUCTURE, DYNAMICS, AND FUNCTION USING NETWORKX
Author(s):	ARIC HAGBERG PIETER SWART DANIEL SCHULT
Intended for:	PROCEEDINGS/TALK SCIPY 08

Los Alamos
NATIONAL LABORATORY
EST. 1943

Form 836 (7/06)

Exploring network structure, dynamics, and function using NetworkX

Aric Hagberg and Pieter J. Swart
Mathematical Modeling and Analysis, Theoretical Division,
Los Alamos National Laboratory, Los Alamos, NM 87545

Daniel A. Schult
Department of Mathematics, Colgate University, Hamilton, NY 13346

NetworkX is a Python language package for exploration and analysis of networks and network algorithms. The core package provides data structures for representing many types of networks, or graphs, including simple graphs, directed graphs, and graphs with parallel edges and self loops. The nodes in NetworkX graphs can be any (hashable) Python object and edges can contain arbitrary data; this flexibility makes NetworkX ideal for representing networks found in many different scientific fields.

In addition to the basic data structures many graph algorithms are implemented for calculating network properties and structure measures: shortest paths, betweenness centrality, clustering, and degree distribution and many more. NetworkX can read and write various graph formats for easy exchange with existing data, and provides generators for many classic graphs and popular graph models, such as the Erdös-Rényi, Small World, and Barabási-Albert models, are included.

The ease-of-use and flexibility of the Python programming language together with connection to the SciPy tools make NetworkX a powerful tool for scientific computations. We discuss some of our recent work studying synchronization of coupled oscillators to demonstrate how NetworkX enables research in the field of computational networks.

I. INTRODUCTION

Recent major advances in the theory of networks combined with the ability to collect large-scale network data has increased interest in exploring and analyzing large networks [New03] [BNFT04]. Applications of network analysis techniques are found in many scientific and technological research areas such as gene expression and protein interaction networks, Web Graph structure, Internet traffic analysis, social and collaborative networks including contact networks for the spread of diseases. In these areas and others, specialized software tools are available that solve domain-specific computational problems but only recently have open-source general purpose tools been developed that can span research application domains [CN] [OFS08]. We have designed NetworkX to fill the need for general network analysis software that also can be easily used as a platform for designing new theory and algorithms [HSS].

The NetworkX package is a flexible network analysis tool written in the Python programming language. NetworkX provides basic network, or graph, data structures that allow the representation of simple graphs, directed graphs, and graphs with self-loops and parallel edges. It allows (almost) arbitrary objects as nodes and can associate arbitrary objects to edges. This means that the network structure can be integrated with custom objects and data structures, complementing any pre-existing code and allowing network analysis in any application setting without significant software development. Once a network is represented as a NetworkX object, standard algorithms that facilitate finding degree distributions (number of edges incident to each node), clustering coefficients (number of triangles each node is part of), shortest paths, spectral measures, and communities can be used to analyze the structure.

We began developing NetworkX in 2002 to analyze data and intervention strategies for the epidemic spread of disease [EGK02] and to study the structure and dynamics of social, biological, and infrastructure networks. The initial development was driven by our need for ease-of-use and rapid development in a collaborative, multidisciplinary environment. Our initial goals were to build an open-source tool base that can easily grow in a multidisciplinary environment with users and developers that are not necessarily experts in software architecture or programming. We wanted to build something that interfaces easily with existing code bases written in C, C++, and FORTRAN, and that could painlessly slurp in large nonstandard data sets (one of our early tests involve studying dynamics on a 1.6 million node graph with 6 million edges). Python satisfied all of our requirements but there was no existing API or graph implementation that was suitable for our project. Inspired by a 1998 essay by Python creator Guido van Rossum on a Python graph representation [vR98] and the excellent C and C++ graph data structures and algorithms book by Sedgewick [Sed02] we developed NetworkX as a tool for the field of computational networks. NetworkX had a public premier in September 2004 at the SciPy annual conference and was first publicly released in April of 2005.

In this paper we describe NetworkX and demonstrate how it has enabled our work studying synchronization of coupled oscillators. In the following we give a brief introduction to NetworkX with simple examples and describe some of the details of the classes, data structures and algorithms available. After that we describe in detail a research project in which NetworkX plays a central role. We conclude with examples of how others have used NetworkX in research and education.

II. USING NETWORKX

To get started with NetworkX you will need the Python language system and the NetworkX package. Both are included in several standard operating system packages [pac]. NetworkX is easy to install and we suggest you visit the project website to make sure you have the latest software version and documentation [HSS]. In some of the following examples we also show how NetworkX interacts with other optional Python packages such as NumPy, SciPy, and Matplotlib, and we suggest you also consider installing those; NetworkX will automatically use them if they are available.

To get started first import NetworkX using "nx" as a short name to save typing

```
>>> import networkx as nx
```

The basic *Graph* class is used to hold the network information. Nodes can be added as follows:

```
>>> G=nx.Graph()
>>> G.add_node(1) # integer
>>> G.add_node('a') # string
>>> print G.nodes()
['a', 1]
```

Nodes can be any hashable [has] object such as strings, numbers, files, functions, and more

```
>>> import math
>>> G.add_node(math.cos) # cosine function
>>> fh=open('tmp.txt','w')
>>> G.add_node(fh) # file handle
>>> print G.nodes()
[<built-in function cos>,
<open file 'tmp.txt', mode 'w' at 0x30dc38>]
```

Edges, or links, between nodes are represented as tuples of nodes. They can be added simply

```
>>> G.add_edge(1,'a')
>>> G.add_edge('b',math.cos)
>>> print G.edges()
[('b', <built-in function cos>), ('a', 1)]
```

If the nodes do not already exist they are automatically added to the graph.

Edge data can d be associated with the edge by adding an edge as a 3-tuple (u, v, d). The default value for d is the integer 1 but any valid Python object is allowed. Using numbers as edge data allows a natural way to express weighted networks. In the following example we use Dijkstra's algorithm to find the shortest weighted path through a simple network of four edges with weights.

```
>>> G=Graph()
>>> e=[('a','b',0.3),('b','c',0.9),
       ('a','c',0.5),('c','d',1.2)]
>>> G.add_edges_from(e)
>>> print dijsktra_path(G,'a','d')
[0, 2, 3]
```

NetworkX includes functions for computing network statistics and metrics such as diameter, degree distribution, number of connected components, clustering coefficient, and betweenness centrality. In addition generators for many classic graphs and random graph models are provided. These graphs are useful for modeling and analysis of network data and also for testing new algorithms or network metrics. The following example shows how to generate a network consisting of a path with 6 nodes and compute some statistics about that network.

```
>>> G = nx.path_graph(6)
>>> print G.degree()
[1, 2, 2, 2, 2, 1]
>>> print nx.density(G)
0.333333333333
>>> print nx.diameter(G)
5
>>> print nx.degree_histogram(G)
[0, 2, 4]
>>> print nx.betweenness_centrality(G)
{0: 0.0, 1: 0.4, 2: 0.6, 3: 0.6, 4: 0.4, 5: 0.0}
```

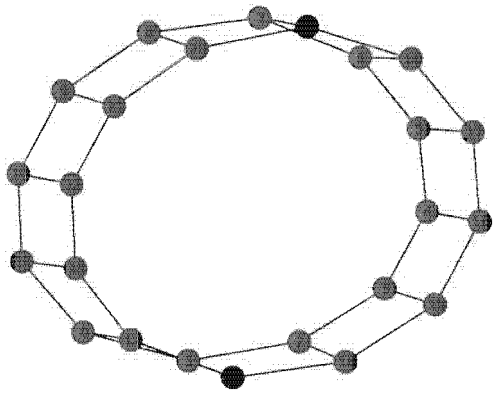

FIG. 1: Matplotlib plot of a 24 node circular ladder graph

NetworkX leverages existing Python libraries to extend the available functionality with interfaces to well-tested numerical and statistical libraries written in C, C++ and FORTRAN. NetworkX graphs can easily be convert to NumPy matrices and SciPy sparse matrices to leverage the linear algebra, statistics, and other tools from those packages. For example, to study the eigenvalue spectrum of the graph Laplacian the NetworkX *laplacian()* function returns a NumPy matrix representation. The eigenvalues can be then easily computed using the numpy.linalg sub-package

```
>>> L=nx.laplacian(G)
>>> print L  # a NumPy matrix
[[ 1. -1.  0.  0.  0.  0.]
 [-1.  2. -1.  0.  0.  0.]
 [ 0. -1.  2. -1.  0.  0.]
 [ 0.  0. -1.  2. -1.  0.]
 [ 0.  0.  0. -1.  2. -1.]
 [ 0.  0.  0.  0. -1.  1.]]
>>> import numpy.linalg
>>> print numpy.linalg.eigvals(L)
[  3.7321e+00   3.0000e+00   2.0000e+00
   1.0000e+00  -4.0235e-17   2.6795e-01]
```

For visualizing networks NetworkX includes an interface to the the Matplotlib Python plotting package along with simple force-directed, spectral, and geometric node positioning algorithms.

```
>>> G = nx.circular_ladder_graph(12)
>>> nx.draw(G)
```

Connections to other graph drawing packages are available either directly, for example using PyGraphviz with Graphviz drawing system, or through writing the data to one of the standard file interchange formats.

III. INSIDE NETWORKX

NetworkX provides classes to represent directed and undirected graphs, with optional weights and self loops, and a special representation for multigraphs which allows multiple edges between pairs of nodes. Basic graph manipulations such as adding more removing nodes or edges are provided class methods. Some standard graph

reporting such as listing nodes or edges or computing node degree are also provided as class methods but more complex statistics and algorithms such as clustering, shortest paths, and visualization are provided as package functions.

There are a few standard representations for graphs: a list of edges, an adjacency matrix, or an adjacency list. The choice of representation affects both the storage and computational time to perform look-ups and graph algorithms [Sed02]. Many real-world graphs and network models are sparse; they typically have only a few connections of the total possible connections for each node. For sparse graphs the adjacency list is preferred since the storage requirement is the smallest (proportional to $m + n$ for n nodes and m edges).

There are several options for implementing an adjacency list in Python using the built-in list, set, and dictionary data structures. The simplest option is to use a "dictionary of lists"[vR98] where each node v is a key in a dictionary with associated data consisting of a list of neighbors (nodes connected to v). Another possibility is to use a "dictionary of sets" by using a set of nodes instead of a list.

NetworkX in contrast uses a "dictionary of dictionaries". The representation of an undirected graph with the edges $A - B$, $B - C$, is

```
>>> G=networkx.Graph()
>>> G.add_edge('A','B')
>>> G.add_edge('B','C')
>>> print G.adj
{'A': {'B': 1},
 'B': {'A': 1, 'C': 1},
 'C': {'B': 1}}
```

Each node n is a key in the `G.adj` dictionary with the data consisting of a dictionary with neighbors as keys and the default data 1 as value. This node dictionary allows the natural expressions `n in G` to test if the graph G contains node n and and `for n in G` to loop over all nodes [Epp08]. The "dictionary of dictionary" data structure allows finding edges and removing edges with two dictionary look-ups instead of a dictionary look-up and a search when using a "dictionary of lists". Some of the same benefits can be realized using sets to represent the node neighbors but we use dictionaries instead since this allows arbitrary data to be attached to the edge. The phrase `G[u][v]` returns the edge object associated with the edge between nodes u and v. A common use is to store a real value on the edge so that a weighted graph is represented.

Note that for undirected graphs both representations (e.g $A - B$ and $B - A$) are stored. Storing both representations allows a single dictionary look-up to test if edge $u - v$ or $v - u$ exists. For directed graphs only one of the representations for the edge $u \rightarrow v$ needs to be stored but we keep track of both the forward edge and the backward edge in a "successor" and "predecessor" dictionary of dictionaries. This extra storage simplifies some algorithms, such as finding shortest paths, when traversing backwards through a graph is useful.

Though less natural than storing simple graphs or digraph the "dictionary of dictionaries" data structure can also be used to store graphs with parallel edges (multigraphs). NetworkX provides the $MultiGraph$ and $MultiDiGraph$ classes to implement a graph structure with parallel edges. In that case the data for $G[u][v]$ consists of a list of edge objects with one element for each edge connecting nodes u and v.

There are no custom node objects or edge objects by default in NetworkX. Edges in NetworkX are represented as a two-tuple or three-tuple of nodes (u, v), or (u, v, d) with d as edge data. The edge data d is the value of a dictionary and can thus be any Python object. Nodes are keys in a dictionary and therefore have the same restrictions as Python dictionaries; nodes must be hashable objects. Users can define custom node objects as long as they meet that single requirement.

IV. NETWORKX IN ACTION: SYNCHRONIZATION

We are using NetworkX in our scientific research for the spectral analysis of network dynamics and to study synchronization in networks of coupled oscillators [HS08]. Synchronization of oscillators is a fundamental problem of dynamical systems with applications to heart and muscle tissue, ecosystem dynamics, secure communication with chaos, neural coordination, memory and epilepsy. The specific question we are investigating is how to best rewire a network in order to enhance or decrease the network's ability to synchronize. We are particularly interested in the setting where the number of edges in a network stays the same; we can modify the network by moving edges (defined as removing an edge between one pair of nodes and adding an edge between another). Ideally this question should be answered independently of the specific details of the oscillators or coupling.

Our model follows the framework presented by [FJC00] where identical oscillators are coupled in a fairly general manner and said to be synchronized if their states are identical at all times. Small perturbations from synchronization are examined to determine if they grow or decay. If the perturbations decay the system is said to be synchronizable. In solving for the growth rate of perturbations, it becomes apparent that the dynamical characteristics of the oscillator and coupling separate from the structural properties of the network over which they are coupled. This surprising and powerful separation implies that coupled oscillators synchronize more effectively on certain networks independent of the type of oscillator or their coupling.

The effect of the network structure on synchronization is determined via the eigenvalues of the network Laplacian matrix $L = D - A$ where A is the adjacency matrix representation of the network and D is a diagonal matrix of node degrees. For a network with N oscillators, there are N eigenvalues-all real and non-negative. The lowest $\lambda_0 = 0$ is always zero and we index the others λ_i in increasing order. For a connected network it is true that $\lambda_i > 0$ for $i > 0$. The growth rate of perturbations is determined by a Master Stability Function (MSF) which takes eigenvalues as inputs and returns the growth rate for that eigenmode. The observed growth rate of the system is the maximum of the MSF evaluations for all eigenvalues. By studying the spectrum one can show that networks for which the eigenvalues lie in a wide band are resistant to synchronization and an effective measure of the resistance to synchronization is the ratio of the largest to smallest positive eigenvalue of the network, $r = \lambda_{N-1}/\lambda_1$ The goal of enhancing synchronization is then to move edges that optimally decrease r.

Python makes it easy to implement algorithms quickly and test how well they work. Functions that take NetworkX.Graph() objects as input and return an edge constitute an algorithm for edge addition or removal. Combining these gives algorithms for moving edges. We implemented several algorithms using either the degree of each node or the eigenvectors of the network Laplacian and compared their effectiveness to each other and to random edge choice. We found that while algorithms which use degree information are much better than random edge choice, it is most effective to use information from the eigenvectors of the network rather than degree.

Of course, the specific edge to choose for rewiring depends on the network you start with. NetworkX is helpful for exploring edge choices over many different networks since a variety of networks can be easily created. Real data sets that provide network config-

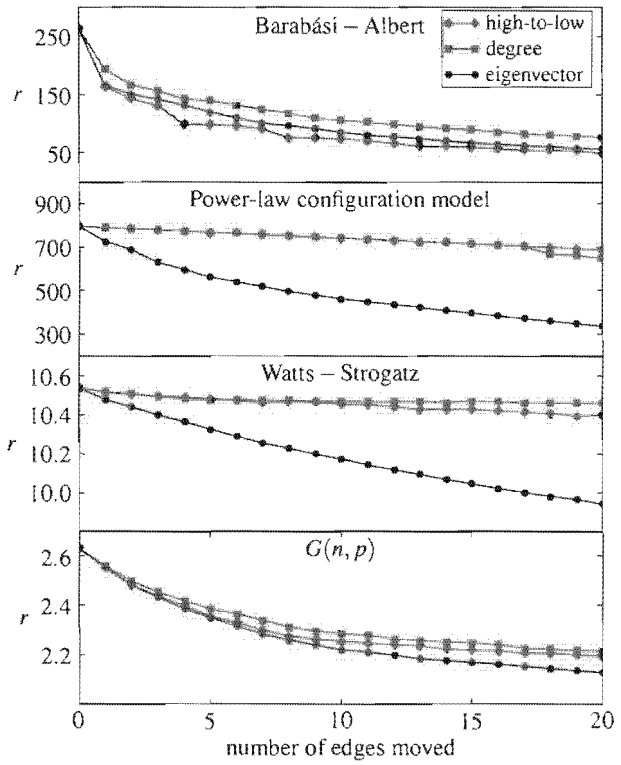

FIG. 2: The change in r as edges in some example networks are moved according to different schemes. A greedy strategy that moves edges based on Laplacian eigenvectors is the most effective overall at enhancing synchronization by reducing r. The Laplacian eigenvalues found by using the NetworkX connections to SciPy and NumPy matrix eigenvalue solvers.

urations can be read into Python using simple edge lists as well as many other formats. In addition, a large collection of network model generators are included so that, for example, random networks with a given degree distribution can be easily constructed. These generator algorithms are taken from recent (as well as very old) literature on random network models. The Numpy package also makes it easy to collect statistics over many networks and plot the results via Matplotlib as shown in Fig. 2.

In addition to computation, visualization of the networks is helpful. NetworkX hooks into Matplotlib or Graphviz (2D) and VTK or UbiGraph (3D) allow network visualization with node and edge traits that correlate well with r as shown in Fig. 3.

V. NETWORKX IN THE WORLD

The core of NetworkX is written completely in Python; this makes the code easy to read, write, and document. Using Python lowers the barrier for students and non-experts to learn, use, and develop network algorithms. The ease-of-use has contributed to uses in the open-source community and in university educational settings [MS07]. The SAGE open source mathematics system [Ste08] has incorporated NetworkX and extended it with even more graph-theoretical algorithms and functions.

NetworkX takes advantage of many existing applications in

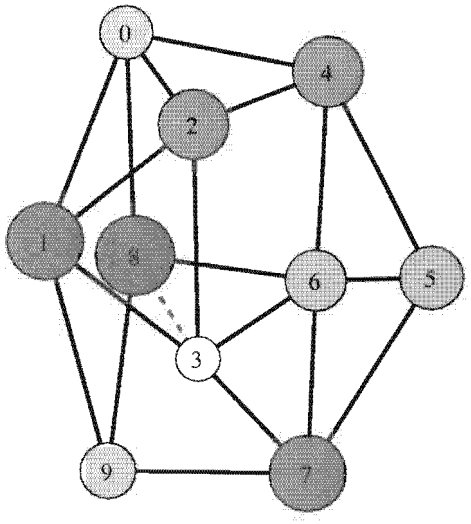

FIG. 3: A sample graph showing how to choose edges by eigenvector. The size of each node represents the value of the largest eigenvector associated with that node. The dashed edge is the edge with the largest difference in eigenvector values between the two nodes. Nodes 3 and 6 have the highest degree but the edge between 3 and 8 is more effective at enhancing synchronization.

Python and other languages and brings then together to build a powerful analysis platform. For computational analysis of networks using techniques from algebraic graph theory, NetworkX uses adjacency matrix representations of networks with NumPy dense matrices and SciPy sparse matrices [Oli06]. The NumPy and SciPy packages also provide linear system and eigenvalue solvers, statistical tools, and many more useful functions. For visualizing and drawing, NetworkX contains interfaces to the Graphviz network layout tools [EGK04], Matplotlib (2d) [Hun07] and UbiGraph (3d) [Vel07]. A variety of standard network Models are included for realization and creation of network models and NetworkX can import graph data from many external formats.

VI. CONCLUSION

Python provides many tools to ease exploration of scientific problems. One of its strengths is the ability connect existing code and libraries in a natural way that eases integration of many tools. Here we have shown NetworkX, in conjunction with packages SciPy, NumPy, Matplotlib and their connections to LINPACK, ODE integration tools and other tools written in FORTRAN and C allow analysis and implementation of algorithms for analyzing dynamics of network coupled oscillators. We hope to have enticed you to take a look at NetworkX the next time you need a way to keep track of connections between objects.

[BNFT04] Eli Ben-Naim, Hans Frauenfelder, and Zoltan Torozckai, editors. *Complex Networks*, volume 650 of *Lecture Notes in Physics*. Springer, 2004.

[CN] Gábor Csárdi and Tamás Nepusz. The igraph library. http://cneurocvs.rmki.kfki.hu/igraph/.

[EGK02] Stephen Eubank, Hasan Guclu, V. S. Anil Kumar, Madhav V. Marathe, Aravind Srinivasan, Zoltan Toroczkai, and Nan Wang. Modelling disease outbreaks in realistic urban social networks. *Nature*, page 180, 2002.

[EGK04] J. Ellson, E.R. Gansner, E. Koutsofios, S.C. North, and G. Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2004.

[Epp08] David Eppstein. PADS, a library of Python Algorithms and Data Structures, 2008. http://www.ics.uci.edu/ eppstein/PADS/.

[FJC00] Kenneth S. Fink, Gregg Johnson, Tom Carroll, Doug Mar, and Lou Pecora. Three coupled oscillators as a universal probe of synchronization stability in coupled oscillator arrays. *Phys. Rev. E*, 61(5):5080 – 90, MAY 2000.

[has] Hashable note.

[HS08] Aric Hagberg and Daniel A. Schult. Rewiring networks for synchronization. To appear in *Chaos*, 2008.

[HSS] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. NetworkX. `https://networkx.lanl.gov`.

[Hun07] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science and Engineering*, 9(3):90–95, May/June 2007.

[MS07] Christopher R. Myers and James P. Sethna. Python for education: Computational methods for nonlinear systems. *Computing in Science and Engineering*, 9(3):75–79, 2007.

[New03] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167 – 256, June 2003.

[OFS08] Joshua O'Madadhain, Danyel Fisher, Padhraic Smyth, Scott White, and Yan-Biao Boey. Analysis and visualization of network data using JUNG. http://jung.sourceforge.net/doc/JUNG_journal.pdf, 2008.

[Oli06] Travis E. Oliphant. *Guide to NumPy*. Provo, UT, March 2006.

[pac] Available in Debian Linux and Fink (OSX) package systems.

[Sed02] Robert Sedgewick. *Algorithms in C: Part 5: Graph algorithms*. Addison-Wesley, Reading, MA, USA, third edition, 2002.

[Ste08] William Stein. *Sage: Open Source Mathematical Software (Version 2.10.2)*. The Sage Group, 2008. `http://www.sagemath.org`.

[Vel07] Todd L. Veldhuizen. Dynamic multilevel graph visualization. Eprint arXiv:cs.GR/07121549, Dec 2007.

[vR98] Guido van Rossum. Python Patterns - Implementing Graphs, 1998. http://www.python.org/doc/essays/graphs/.