Exploring Network Structure, Dynamics, and Function using NetworkX

Aric A. Hagberg (hagberg@lanl.gov) – *Los Alamos National Laboratory, Los Alamos, New Mexico* USA
Daniel A. Schult (dschult@colgate.edu) – *Colgate University, Hamilton, NY* USA
Pieter J. Swart (swart@lanl.gov) – *Los Alamos National Laboratory, Los Alamos, New Mexico* USA

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 Erdos-Renyi, Small World, and Barabasi-Albert models.

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.

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. The rapid growth in network theory has been fueled by its multidisciplinary impact; it provides an important tool in a systems approach to the understanding of many complex systems, especially in the biological sciences.

In these research areas and others, specialized software tools are available that solve domain-specific problems but there are few open-source general-purpose computational network tools [CN] [OFS08]. NetworkX was developed in response to the need for a well-tested and well-documented, open source network analysis tool that can easily span research application domains. It has effectively served as a platform to design theory and algorithms, to rapidly test new hypotheses and models, and to teach the theory of networks.

The structure of a network, or graph, is encoded in the edges (connections, links, ties, arcs, bonds) between nodes (vertices, sites, actors). NetworkX provides basic network data structures for 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 is a powerful advantage; 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, the network structure can be analyzed using standard algorithms for 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.

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 rapid development in a collaborative, multidisciplinary environment. Our initial goals were to build an open-source tool base that could easily grow in a multidisciplinary environment with users and developers that were not necessarily experts in programming or software engineering. We wanted to interface easily with existing code written in C, C++, and FORTRAN, and to painlessly slurp in large nonstandard data sets (one of our early tests involve studying dynamics on a 1.6 million node graph with roughly 10 million edges that were changing with time). 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] we developed NetworkX as a tool for the field of computational networks. NetworkX had a public premier at the 2004 SciPy annual conference and was released as open source software in April 2005.

In this paper we describe NetworkX and demonstrate how it has enabled our recent work studying synchronization of coupled oscillators. In the following we give a brief introduction to NetworkX with basic examples that demonstrate some of the classes, data structures, and algorithms. 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.

A. Hagberg, D. Schult, P. Swart: Proc. SciPy 2008, G. Varoquaux, T. Vaught, J. Millman (Eds), pp. 11–16

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.

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

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

Nodes can be any hashable object such as strings, numbers, files, or functions,

```
>>> 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)]
```

When adding an edge, if the nodes do not already exist they are automatically added to the graph.

Edge data d can 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 = networkx.Graph()
>>> e = [('a','b',0.3),('b','c',0.9),
         ('a','c',0.5),('c','d',1.2)]
>>> G.add_edges_from(e)
>>> print networkx.dijsktra_path(G,'a','d')
['a', 'c', 'd']
```

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 and compute some statistics for a network consisting of a path with 6 nodes:

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

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 converted 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 = networkx.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 Python's Matplotlib plotting package along with simple node positioning algorithms based on force-directed, spectral, and geometric methods.

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

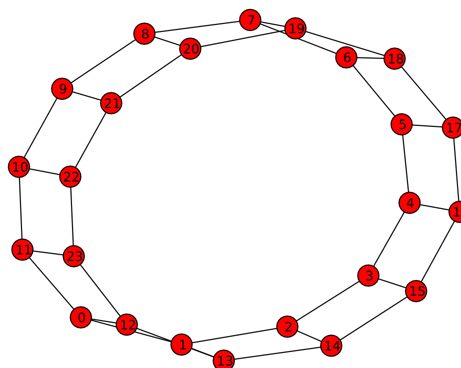

Figure 1: Matplotlib plot of a 24 node circular ladder graph

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

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 or removing nodes or edges are provided as 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.

The standard data structures for representating graphs are edge lists, adjacency matrices, and adjacency lists. The choice of data structure affects both the storage and computational time for graph algorithms [Sed02]. For large sparse networks, in which only a small fraction of the possible edges are present, adjacency lists are preferred since the storage requirement is the smallest (proportional to $m + n$ for n nodes and m edges). Many real-world graphs and network models are sparse so NetworkX uses adjacency lists.

Python built-in dictionaries provide a natural data structure to search and update adjacency lists [vR98]; NetworkX uses a "dictionary of dictionaries" ("hash of hashes") as the basic graph data structure. Each node n is a key in the $G.adj$ dictionary with value consisting of a dictionary with neighbors as keys to edge data values with default 1. For example, the representation of an undirected graph with edges $A-B$ and $B-C$ 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}}
```

The outer node dictionary allows the natural expressions *n in G* to test if the graph G contains node n and *for n in G* to loop over all nodes [Epp08]. The "dictionary of dictionary" data structure allows finding and removing edges with two dictionary look-ups instead of a dictionary look-up and a search when using a "dictionary of lists". The same fast look-up could be achieved using sets of neighbors, but neighbor dictionaries allow arbitrary data to be attached to an edge; the phrase $G[u][v]$ returns the edge object associated with the edge between nodes u and v. A common use is to represent a weighted graph by storing a real number value on the edge.

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 $u \to v$ needs to be stored but we keep track of both the forward edge and the backward edge in distinct "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.

The "dictionary of dictionaries" data structure can also be used to store graphs with parallel edges (multigraphs) where the data for $G[u][v]$ consists of a list of edge objects with one element for each edge connecting nodes u and v. NetworkX provides the $MultiGraph$ and $MultiDiGraph$ classes to implement a graph structure with parallel edges.

There are no custom node objects or edge objects by default in NetworkX. Edges 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. Users can define arbitrary custom edge objects.

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 while modifying the network by moving edges (defined as removing an edge between one pair of nodes and adding an edge between another). What are the network properties that seriously diminish or enhance synchronization and how hard is it to calculate the required rewirings?

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 form of 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 which are 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. The separation comes about because the MSF is determined by the oscillator and coupling but not by the network structure which only impacts the inputs to the MSF. So long as all eigenvalues lie in an interval where the MSF is negative, the network is synchronizable. Since most oscillator/couplings lead to MSFs where a single interval yields negative growth rates, networks for which the eigenvalues lie in a wide band are resistant to synchronization. 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.

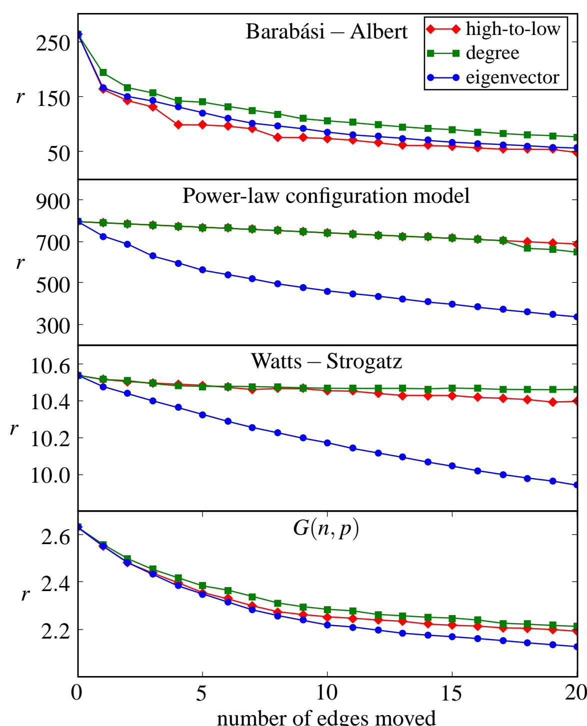

Figure 2: The change in resistance to synchrony r as edges are moved in four example random network models. An algorithm using Laplacian eigenvectors compares favorably to those using node degree. Eigenvectors are found via NetworkX calls to SciPy and NumPy matrix eigenvalue solvers.

Python makes it easy to implement such 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 configurations 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 the literature on random network models. The Numpy package 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 provide hooks into Matplotlib or Graphviz (2D) and VTK or UbiGraph (3D) and thereby allow network visualization with node and edge traits that correlate well with r as shown in Fig. 3.

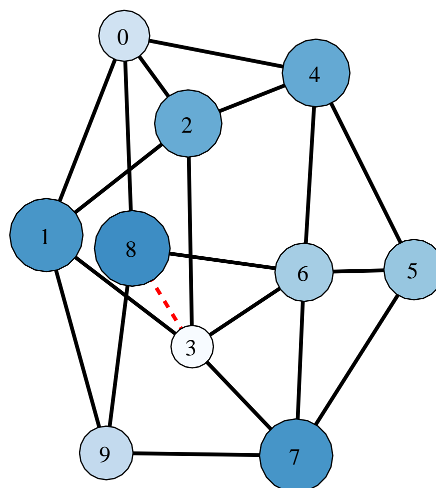

Figure 3: A sample graph showing eigenvector elements associated with each node as their size. The dashed edge shows the largest difference between two nodes. Moving the edge between nodes 3 and 8 is more effective at enhancing synchronization than the edge between the highest degree nodes 3 and 6.

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 low barrier has encouraged contributions from 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 Python and other languages and brings then together to build a powerful analysis platform. For the 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 other 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.

Conclusion

Python provides many tools to ease exploration of scientific problems. One of its strengths is the ability to connect existing code and libraries in a natural way that eases integration of many tools. Here we have shown how NetworkX, in conjunction with the Python packages SciPy, NumPy, Matplotlib and connection to other tools written in FORTRAN and C, provides a powerful tool for computational network analysis. 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.

Acknowledgements

As an open source project this work has significantly benefited from its own international social network of users. We thank the user community for feedback, bug reports, software contributions and encouragement. This work was carried out under the auspices of the National Nuclear Security Administration of the U.S. Department of Energy at Los Alamos National Laboratory under Contract No. DE-AC52-06NA25396 and partially supported by the Laboratory Directed Research and Development Program.

References

[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.

[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/