NetworkX
Exploring network structure, dynamics, and function

Aric Hagberg\textsuperscript{1}  Daniel Schult\textsuperscript{2}  Pieter Swart\textsuperscript{1}

\textsuperscript{1}Theoretical Division,
Los Alamos National Laboratory, Los Alamos, New Mexico 87545, USA

\textsuperscript{2}Department of Mathematics, Colgate University, Hamilton, NY 13346, USA

5 March 2009
Vast amounts of data are being generated and collected

- **Technology:** Internet, telecommunications, power grid
- **Sociology:** WWW, email, social networking
- **Biology:** protein interactions, genetic regulatory networks, epidemiology

Need theory, analysis, models
Example: social networks and epidemics

Understand epidemic outbreak of diseases through modeling
Build social networks from detailed census data
Run dynamic models for smallpox, SARS, flu, etc.

Building a social network

Goal: find a good intervention strategy
NISAC: EpiSimS

Social network of one person
Example: interdiction

Problem: smuggling of nuclear material in transportation network

Potential source site (Kurchatov Institute, Moscow).

Detector at border crossing

Image credits: NTI
Example: interdiction

Find best set of roads (edges) to monitor (cut) with limited budget
University libraries, journals, and aggregators collect journal usage data through web portals. 
MESUR project is analyzing about 1 billion usage events. 
Build network from user click streams.

- Do scholars read and cite journals in the same way?
- Can new trends in research (new field, interdisciplinary) be spotted?
- Which journals are most important according to usage?

Johan Bollen, Los Alamos
Example: journal usage network
## Why we started project

### We needed:
- Tool to study the structure and dynamics of social, biological, and infrastructure networks
- Ease-of-use and rapid development in a collaborative, multidisciplinary environment
- Open-source tool base that can easily grow in a multidisciplinary environment with non-expert users and developers
- An easy interface to existing code bases written in C, C++, and FORTRAN
- To painlessly slurp in large nonstandard data sets

- No existing API or graph implementation that was suitable
- Inspired by Guido van Rossum’s 1998 Python graph representation essay
- First public release in April 2005
NetworkX in one slide

- Python language package for exploration and analysis of networks and network algorithms
- Data structures for representing many types of networks, or graphs, (simple graphs, directed graphs, and graphs with parallel edges and self loops)
- Nodes can be any (hashable) Python object
- Edges can contain arbitrary data
- Flexibility ideal for representing networks found in many different fields
Using NetworkX
Adding nodes

Start Python
Import NetworkX using “nx” as a short name

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

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

```python
>>> G=nx.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, functions, and more

```python
>>> import math
>>> G.add_node(math.cos)  # cosine function
>>> fh=open('tmp.txt','w')
>>> G.add_node(fh)  # file handle
>>> 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:

```python
>>> 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 $d$ assigned using a 3-tuple $(u, v, d)$
Default $d$ is (integer) 1 - any Python object is allowed

Use Dijkstra’s algorithm to find the shortest weighted path:

```python
>>> 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')
['a', 'c', 'd']
```
Graph generators and statistics

Generators for many classic graphs and random graph models
Used for modeling and testing new algorithms
Generate 6 node path and compute some statistics

```python
>>> 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}
```
It’s “Python all the way down”
NetworkX uses a “dictionary of dictionaries”
Good for adjacency list representation (sparse graphs)

- Node $n$ is a key in the $G$.adj dictionary
- Data is a dictionary with neighbors as keys and data

Representation of an undirected graph with the edges $A \rightarrow B$, $B \rightarrow C$

```python
>>> G=nx.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}}
```
Guido van Rossum proposed a dictionary of lists
Allows the natural expressions (Eppstein)

- “n in G” to test if the graph G contains node n
- “for n in G” to loop over all nodes

Advantages of “dict of dict” data structure

- Find edges and remove edges with two dictionary look-ups
- Prefer to “sets” since data can be attached to edge
  - $G[u][v]$ returns the edge object


Design decisions

**NetworkX defines no custom node objects or edge objects**
- “node-centric” view of network
- Nodes: whatever you put in (hashable)
- Edges: tuples with optional edge data (three-tuple)
- Edge data is arbitrary and users can define custom node types

**NetworkX is all Python**
(Other projects use custom compiled code and Python: Boost Graph, igraph, Graphviz)
- Focus on computational network modeling not software tool development
- Move fast to design new algorithms or models
Writing a simple algorithm

**Breadth First Search**

```python
from collections import deque

def bfs(g, source):
    queue = deque([(None, source)])
    enqueued = set([source])
    while queue:
        parent, n = queue.popleft()
        yield parent, n
        new = set(g[n]) - enqueued
        enqueued |= new
        queue.extend([(n, child) for child in new])

Credit: Matteo Dell’Amico
```
def shortest_path(g, source, target):
    paths = {None: []}
    for parent, child in bfs(g, source):
        paths[child] = paths[parent] + [child]
        if child == target:
            return paths[child]
    return None  # or raise appropriate exception

Credit: Matteo Dell’Amico
Directed Scale-Free Graphs

Béla Bollobás\textsuperscript{*} Christian Borgs\textsuperscript{†} Jennifer Chayes\textsuperscript{†} Oliver Riordan\textsuperscript{§}

2 The model
We consider a directed graph which grows by adding single edges at discrete time steps. At each such step a vertex may or may not also be added. For simplicity we allow multiple edges and loops. More precisely, let \( \alpha, \beta, \gamma, \delta_{in} \) and \( \delta_{out} \) be non-negative real numbers, with \( \alpha + \beta + \gamma = 1 \). Let \( G_0 \) be any fixed initial directed graph, for example a single vertex without edges, and let \( t_0 \) be the number of edges of \( G_0 \). (Depending on the parameters, we may have to assume \( t_0 \geq 1 \) for the first few steps of our process to make sense.) We set \( G(t_0) = G_0 \), so at time \( t \) the graph \( G(t) \) has exactly \( t \) edges, and a random number \( n(t) \) of vertices. In what follows, to choose a vertex \( v \) of \( G(t) \) according to \( d_{out} + \delta_{out} \) means to choose \( v \) so that \( \Pr(v = v_i) \) is proportional to \( d_{out}(v_i) + \delta_{out} \), i.e., so that \( \Pr(v = v_i) = (d_{out}(v_i) + \delta_{out})/(t + \delta_{out} n(t)) \). To choose \( v \) according to \( d_{in} + \delta_{in} \) means to choose \( v \) so that \( \Pr(v = v_i) = (d_{in}(v_i) + \delta_{in})/(t + \delta_{in} n(t)) \). Here \( d_{out}(v_i) \) and \( d_{in}(v_i) \) are the out-degree and in-degree of \( v_i \), measured in the graph \( G(t) \).

For \( t \geq t_0 \) we form \( G(t+1) \) from \( G(t) \) according the following rules:

(A) With probability \( \alpha \), add a new vertex \( v \) together with an edge from \( v \) to an existing vertex \( w \), where \( w \) is chosen according to \( d_{in} + \delta_{in} \).

(B) With probability \( \beta \), add an edge from an existing vertex \( v \) to an existing vertex \( w \), where \( v \) and \( w \) are chosen independently, \( v \) according to \( d_{out} + \delta_{out} \), and \( w \) according to \( d_{in} + \delta_{in} \).

(C) With probability \( \gamma \), add a new vertex \( w \) and an edge from an existing vertex \( v \) to \( w \), where \( v \) is chosen according to \( d_{out} + \delta_{out} \).
import bisect
import random
from networkx import MultiDiGraph

def scale_free_graph(n, alpha=0.41, beta=0.54, delta_in=0.2, delta_out=0):
    def _choose_node(G, distribution, delta):
        cumsum = 0.0
        psum = float(sum(distribution.values())) + float(delta) * len(distribution)
        r = random.random()
        for i in range(0, len(distribution)):
            cumsum += (distribution[i] + delta) / psum
            if r <= cumsum:
                break
        return i

    G = MultiDiGraph()
    G.add_edges_from([(0, 1), (1, 2), (2, 0)])
    gamma = 1 - alpha - beta

    while len(G) < n:
        r = random.random()
        if r < alpha:
            v = len(G)
            w = _choose_node(G, G.in_degree(with_labels=True), delta_in)
        elif r < alpha + beta:
            v = _choose_node(G, G.out_degree(with_labels=True), delta_out)
            w = _choose_node(G, G.in_degree(with_labels=True), delta_in)
        else:
            v = _choose_node(G, G.out_degree(with_labels=True), delta_out)
            w = len(G)
        G.add_edge(v, w)
    return G
Leveraging libraries

Use well-tested numerical and statistical libraries
Convert to NumPy (and SciPy sparse) matrices
Example: Find eigenvalue spectrum of the graph Laplacian

```python
>>> 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  2.6795e-01]
```
Built-in interface to Matplotlib plotting package
Node positioning algorithms based on force-directed, spectral, and geometric methods

```python
>>> G = nx.circular_ladder_graph(12)
>>> nx.draw(G) # Matplotlib under the hood
```
Drawing with Matplotlib

Hagberg

NetworkX
Drawing with other programs

Graphviz

Output to: dot, GML, LEDA, edge list, adjacency list, YAML, sparsegraph6, GraphML

UbiGraph
Movie: Todd Veldhuizen

http://math.lanl.gov/~hagberg/movies/networkx_ubigraph.mov
Where is NetworkX being used?
Adding red edges allows network to synchronize. Edges found by studying network Laplacian spectrum.

http://math.lanl.gov/~hagberg/movies-sync.mp4
INFO/SOCI 485 Computational Methods for Complex Networks (Gueorgi Kossinets)
Physics 7682 / Computing & Information Sciences 6229 (Chris Myers)
"Reality mining" (Nathan Eagle)

Inferring social network structure using cell phone data
e.g. 1.2M phone users in Rwanda
FBI: Cybercrime

Fighting cybercrime: botnets, spam, phishing
http://networkx.lanl.gov/

Currently at networkx-0.99
networkx-1.0 soon (overdue)

Release 1.0

- Refactor classes for simpler expression of weighted graphs and graph attributes
- New features: network metrics (PageRank, HITS, etc.), spanning trees, graph readers and writers, more...
- Performance improvements (drawing, algorithms)
- New documentation (with Sphinx)

NetworkX is a community effort. Thanks!