5 - Developing Algorithms

Drew Conway and Aric Hagberg

June 29, 2010

Outline

- Examples of some simple algorithms
- Writing a new algorithm

Feature: Compact code - building new generators

Directed Scale-Free Graphs

Béla Bollobás* Christian Borgs† Jennifer Chayes‡ Oliver Riordan§

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 α, β, γ, δ_{in} and δ_{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 the following rules:

(A) With probability α, 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 β, 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 γ, add a new vertex w and an edge from an existing vertex v to w, where v is chosen according to $d_{out} + \delta_{out}$.

Feature: Compact code - building new generators

```python
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(),delta_in)
        elif r < alpha+beta:
            v = _choose_node(G, G.out_degree(),delta_out)
            w = _choose_node(G, G.in_degree(),delta_in)
        else:
            v = _choose_node(G, G.out_degree(),delta_out)
            w = len(G)
        G.add_edge(v,w)
    return G
```

Feature: Python expressivity - a simple algorithm

Python is easy to write and read

Breadth First Search

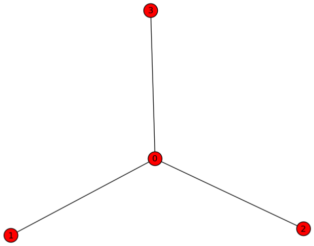
```python
from collections import deque

def breadth_first_search(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

Degree centrality

For a graph with n nodes

$$C_D(v) = \frac{deg(v)}{n-1}$$




```
>>> G=nx.star_graph(3)
>>> print G.edges()
[(0, 1), (0, 2), (0, 3)]
>>> print G.degree(0)
3
>>> print len(G) # # of nodes
4
>>> print G.degree(0)/3
0
>>> print G.degree(0)/3.0
1
>>> for v in G:
...     print v, G.degree(v)/3.0
0 1.0
1 0.333333333333
2 0.333333333333
3 0.333333333333
```

Degree centrality 1

```python
import networkx as nx

def degree_centrality(G):

    n=len(G)-1.0 # forces floating point for n
    for v in G:
        print v,G.degree(v)/n

    return

G=nx.star_graph(3)
degree_centrality(G)
```

Degree centrality 2

```python
import networkx as nx

def degree_centrality(G):

    centrality = {} # empty dictionary
    n=len(G)-1.0 # forces floating point for n
    for v in G:
        centrality[v]=G.degree(v)/n

    return centrality

G=nx.star_graph(3)
dc=degree_centrality(G)
for v in dc:
    print v,dc[v]

print dc
```

```python
def degree_centrality(G):

    centrality = {} # empty dictionary
    n=len(G)-1.0 # forces floating point for n
    for v in G:
        centrality[v]=G.degree(v)/n

    return centrality

if __name__=='__main__':
    import networkx as nx
    G=nx.star_graph(3)
    for v,c in degree_centrality(G).items():
        print v,c
```

```python
def degree_centrality(G):
    """Compute degree centrality for nodes.

    The degree centrality for a node is the fraction of all other
    nodes it is connected to.

    >>> import networkx as nx
    >>> G=nx.star_graph(3)
    >>> print degree_centrality(G)[0]
    1.0
    """
    centrality = {} # empty dictionary
    n=len(G)-1.0 # forces floating point for n
    for v in G:
        centrality[v]=G.degree(v)/n
    return centrality
```

Degree centrality in NetworkX

```python
def degree_centrality(G):
    """Compute the degree centrality for nodes.

    The degree centrality for a node v is the fraction of nodes it
    is connected to.

    Parameters
    ----------
    G : graph
      A networkx graph

    Returns
    -------
    nodes : dictionary
       Dictionary of nodes with degree centrality as the value.

    See Also
    --------
    betweenness_centrality, load_centrality, eigenvector_centrality

    Notes
    -----
    The degree centrality values are normalized by dividing by the maximum
    possible degree in a simple graph n-1 where n is the number of nodes in G.

    For multigraphs or graphs with self loops the maximum degree might
    be higher than n-1 and values of degree centrality greater than 1
    are possible.
    """
    centrality={}
    s=1.0/(len(G)-1.0)
    centrality=dict((n,d*s) for n,d in G.degree_iter())
    return centrality
```

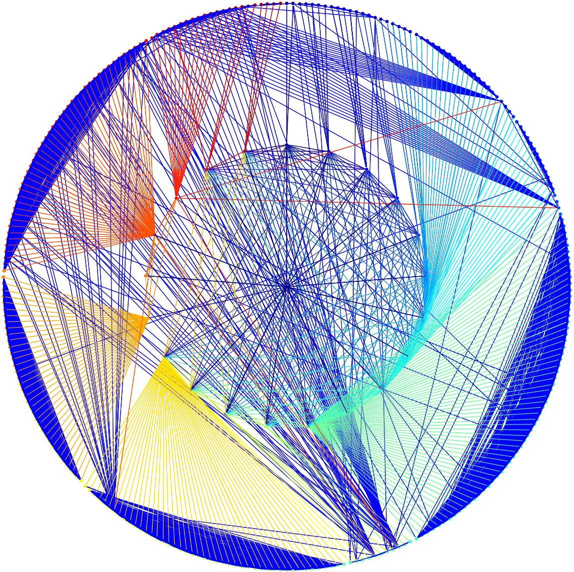
Degree centrality 5

This algorithm is really a one-liner:

```
def degree_centrality(G):
    return dict((n,d/(len(G)-1.0)) for n,d in G.degree_iter())
```

Design Ego Graph

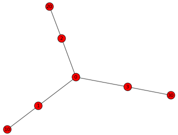
Create network of neighbors centered at node n

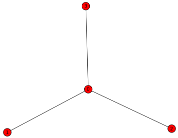


Ego Graph: getting started

```python
import networkx as nx

def ego(G,n):
    """Returns Graph of neighbors centered
    at node n and including n.

    >>> import networkx as nx
    >>> G=nx.star_graph(3)
    >>> G.add_edge(1,10)
    >>> G.add_edge(2,20)
    >>> G.add_edge(3,30)
    >>> E=nx.ego_graph(G,0)
    >>> print E.nodes()
    [0, 1, 2, 3]
    >>> print E.edges()
    [(0, 1), (0, 2), (0, 3)]
    """
    return # E − the ego graph

if __name__=='__main__':
    G=nx.star_graph(3)
    G.add_edges_from([(1,10),(2,20),(3,30)])
    E=ego(G,0)
```

http://bit.ly/cKJhic





Hints:

- only condsider Graph()
- use G.neighbors(v)
- similar as G[v]
- don’t worry about attributes

Ego Network

My solution