5 - Developing Algorithms

Drew Conway and Aric Hagberg

June 29, 2010

- Examples of some simple algorithms
- Writing a new algorithm

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 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

```
 1  import bisect
 2  import random
 3  from networkx import MultiDiGraph
 4
 5  def scale_free_graph(n, alpha=0.41,beta=0.54,delta_in=0.2,delta_out=0):
 6      def _choose_node(G,distribution,delta):
 7          cumsum=0.0
 8          psum=float(sum(distribution.values()))+float(delta)*len(distribution)
 9          r=random.random()
10          for i in range(0,len(distribution)):
11              cumsum+=(distribution[i]+delta)/psum
12              if r < cumsum:
13                  break
14          return i
15
16      G=MultiDiGraph()
17      G.add_edges_from([(0,1),(1,2),(2,0)])
18      gamma=1-alpha-beta
19
20      while len(G)<n:
21          r = random.random()
22          if r < alpha:
23              v = len(G)
24              w = _choose_node(G, G.in_degree(),delta_in)
25          elif r < alpha+beta:
26              v = _choose_node(G, G.out_degree(),delta_out)
27              w = _choose_node(G, G.in_degree(),delta_in)
28          else:
29              v = _choose_node(G, G.out_degree(),delta_out)
30              w = len(G)
31          G.add_edge(v,w)
32      return G
```

Python is easy to write and read

Breadth First Search

```python
1  from collections import deque
2
3  def breadth_first_search(g, source):
4      queue = deque([(None, source)])
5      enqueued = set([source])
6      while queue:
7          parent, n = queue.popleft()
8          yield parent, n
9          new = set(g[n]) - enqueued
10         enqueued |= new
11         queue.extend([(n, child) for child in new])
```

Credit: Matteo Dell'Amico

For a graph with n nodes

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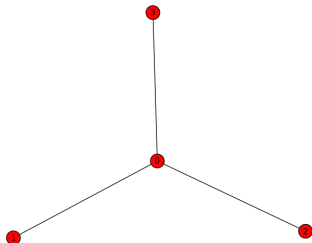

```
1 >>> G=nx.star_graph(3)
2 >>> print G.edges()
3 [(0, 1), (0, 2), (0, 3)]
4 >>> print G.degree(0)
5 2
6 >>> print len(G) # # of nodes
7 4
8 >>> print G.degree(0)/3
9 0
10 >>> print G.degree(0)/3.0
11 1
12 >>> for v in G:
13 ...     print v, G.degree(v)/3.0
14 0 1.0
15 1 0.333333333333
16 2 0.333333333333
17 3 0.333333333333
```

```
1  import networkx as nx
2
3  def degree_centrality(G):
4
5      n=len(G)-1.0 # forces floating point for n
6      for v in G:
7          print v,G.degree(v)/n
8
9      return
10
11 G=nx.star_graph(3)
12 degree_centrality(G)
```

```python
1  import networkx as nx
2
3  def degree_centrality(G):
4
5      centrality = {} # empty dictionary
6      n=len(G)-1.0 # forces floating point for n
7      for v in G:
8          centrality[v]=G.degree(v)/n
9
10     return centrality
11
12 G=nx.star_graph(3)
13 dc=degree_centrality(G)
14 for v in dc:
15     print v,dc[v]
16
17 print dc
```

```
1 def degree_centrality(G):
2
3     centrality = {} # empty dictionary
4     n=len(G)-1.0 # forces floating point for n
5     for v in G:
6         centrality[v]=G.degree(v)/n
7
8     return centrality
9
10 if __name__=='__main__':
11     import networkx as nx
12     G=nx.star_graph(3)
13     for v,c in degree_centrality(G).items():
14         print v,c
```

```python
 1  def degree_centrality(G):
 2      """Compute degree centrality for nodes.
 3
 4      The degree centrality for a node is the fraction of all other
 5      nodes it is connected to.
 6
 7      >>> import networkx as nx
 8      >>> G=nx.star_graph(3)
 9      >>> print degree_centrality(G)[0]
10      1.0
11      """
12      centrality = {} # empty dictionary
13      n=len(G)-1.0 # forces floating point for n
14      for v in G:
15          centrality[v]=G.degree(v)/n
16      return centrality
```

```
1  def degree_centrality(G):
2      """Compute the degree centrality for nodes.
3
4      The degree centrality for a node v is the fraction of nodes it
5      is connected to.
6
7      Parameters
8      _____
9      G : graph
10       A networkx graph
11
12     Returns
13     _____
14     nodes : dictionary
15        Dictionary of nodes with degree centrality as the value.
16
17     See Also
18     _____
19     betweenness_centrality, load_centrality, eigenvector_centrality
20
21     Notes
22     _____
23     The degree centrality values are normalized by dividing by the maximum
24     possible degree in a simple graph n-1 where n is the number of nodes in G.
25
26     For multigraphs or graphs with self loops the maximum degree might
27     be higher than n-1 and values of degree centrality greater than 1
28     are possible.
29     """
30     centrality={}
31     s=1.0/(len(G)-1.0)
32     centrality=dict((n,d*s) for n,d in G.degree_iter())
33     return centrality
```

This algorithm is really a one-liner:

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

Create network of neighbors centered at node n
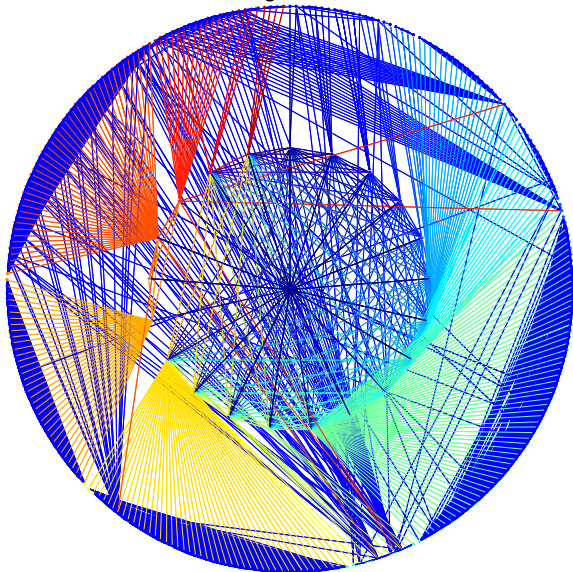
Ego Graph: getting started

```
1  import networkx as nx
2
3  def ego(G,v):
4      """Returns Graph of neighbors centered
5      at node n and including n.
6
7      >>> import networkx as nx
8      >>> G=nx.star_graph(3)
9      >>> G.add_edge(1,10)
10     >>> G.add_edge(2,20)
11     >>> G.add_edge(3,30)
12     >>> E=nx.ego_graph(G,0)
13     >>> print E.nodes()
14     [0, 1, 2, 3]
15     >>> print E.edges()
16     [(0, 1), (0, 2), (0, 3)]
17     """
18     return # E - the ego graph
19
20  if __name__=='__main__':
21      G=nx.star_graph(3)
22      G.add_edges_from([(1,10),(2,20),(3,30)])
23      E=ego(G,0)
```
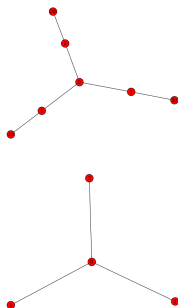
Hints:

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

My solution