In [1]:
```
def pagerank(G, alpha=0.85, personalization=None, max_iter=100, tol=1.0e-6, ns
    if len(G) == 0:
        return {}

    if not G.is_directed():
        D = G.to_directed()
    else:
        D = G

    # Create a copy in (right) stochastic form
    W = nx.stochastic_graph(D, weight=weight)
    N = W.number_of_nodes()

    # Choose fixed starting vector if not given
    if nstart is None:
        x = dict.fromkeys(W, 1.0 / N)
    else:
        # Normalized nstart vector
        s = float(sum(nstart.values()))
        x = dict((k, v / s) for k, v in nstart.items())

    if personalization is None:
        # Assign uniform personalization vector if not given
        p = dict.fromkeys(W, 1.0 / N)
    else:
        missing = set(G) - set(personalization)
        if missing:
            raise NetworkXError('Personalization dictionary must have a value
        s = float(sum(personalization.values()))
        p = dict((k, v / s) for k, v in personalization.items())

    if dangling is None:
        # Use personalization vector if dangling vector not specified
        dangling_weights = p
    else:
        missing = set(G) - set(dangling)
        if missing:
            raise NetworkXError('Dangling node dictionary must have a value fo
        s = float(sum(dangling.values()))
        dangling_weights = dict((k, v/s) for k, v in dangling.items())

    dangling_nodes = [n for n in W if W.out_degree(n, weight=weight) == 0.0]

     # power iteration: make up to max_iter iterations
    for _ in range(max_iter):
        xlast = x
        x = dict.fromkeys(xlast.keys(), 0)
        danglesum = alpha * sum(xlast[n] for n in dangling_nodes)
        for n in x:
            # this matrix multiply looks odd because it is
            # doing a left multiply x^T=xlast^T*W
            for nbr in W[n]:
                x[nbr] += alpha * xlast[n] * W[n][nbr][weight]
```

```
            x[n] += danglesum * dangling_weights[n] + (1.0 - alpha) * p[n]

        # check convergence, l1 norm
        err = sum([abs(x[n] - xlast[n]) for n in x])
        if err < N*tol:
            return x
    raise NetworkXError('Pagerank: power iteration failed to converge in %d it
```

In [2]:
```
import networkx as nx
```

In [3]:
```
G = nx.barabasi_albert_graph(60, 41)
pr = nx.pagerank(G, 0.4)
```

In [4]:
```
print(pr)
```

```
{0: 0.028239119195940958, 1: 0.012774942020766726, 2: 0.01277911955291917, 3:
0.011583329230516855, 4: 0.012977000281773957, 5: 0.012560466982096414, 6: 0.01
3367351679289223, 7: 0.012361127799731066, 8: 0.013378444404555644, 9: 0.013379
47013433502, 10: 0.013361016275672538, 11: 0.012164903923318706, 12: 0.01294575
5859407894, 13: 0.01357621968833211, 14: 0.012957500172750085, 15: 0.0129714954
25634161, 16: 0.01336157185076458, 17: 0.011964580577507548, 18: 0.013777186635
263722, 19: 0.012983285155032808, 20: 0.012969599626064416, 21: 0.0137771866352
63722, 22: 0.012978389807142755, 23: 0.012968605201926654, 24: 0.01195696133502
9983, 25: 0.013565344205968798, 26: 0.012971830186845081, 27: 0.013166601975260
72, 28: 0.012979310425323696, 29: 0.013165412547037339, 30: 0.01255497193685971
8, 31: 0.013160012333167933, 32: 0.013168027798359998, 33: 0.01297526628934456
3, 34: 0.012358537246070726, 35: 0.012742651128325008, 36: 0.01196445947489855
6, 37: 0.01275400672216797, 38: 0.013387981685420443, 39: 0.013573414280059504,
40: 0.013575404211081117, 41: 0.012966821227391942, 42: 0.027826721603092915, 4
3: 0.02750744044230543, 44: 0.027158360191055914, 45: 0.026679403801984136, 46:
0.026153814618654292, 47: 0.02596352412137272, 48: 0.025118776510912287, 49:
0.02498075251844998, 50: 0.024432437580102717, 51: 0.024226711990493935, 52:
0.02365885894933454, 53: 0.023612935275568958, 54: 0.02320433860761948, 55: 0.0
224143387851731, 56: 0.022559852994121603, 57: 0.02200906768766956, 58: 0.02163
4695232599605, 59: 0.021713285964869086}
```

In [ ]:
```

```