Page Rank Algorithm Implementation in Python

```
def pagerank(G, alpha=0.85, personalization=None,
      max_iter=100, tol=1.0e-6, nstart=None, weight='weight',
      dangling=None):
  """Return the PageRank of the nodes in the graph.

  PageRank computes a ranking of the nodes in the graph G based on
  the structure of the incoming links. It was originally designed as
  an algorithm to rank web pages.

  Parameters
  ----------
  G : graph
  A NetworkX graph. Undirected graphs will be converted to a directed
  graph with two directed edges for each undirected edge.

  alpha : float, optional
  Damping parameter for PageRank, default=0.85.

  personalization: dict, optional
  The "personalization vector" consisting of a dictionary with a
  key for every graph node and nonzero personalization value for each node.
  By default, a uniform distribution is used.

  max_iter : integer, optional
  Maximum number of iterations in power method eigenvalue solver.

  tol : float, optional
  Error tolerance used to check convergence in power method solver.

  nstart : dictionary, optional
  Starting value of PageRank iteration for each node.

  weight : key, optional
  Edge data key to use as weight. If None weights are set to 1.

  dangling: dict, optional
  The outedges to be assigned to any "dangling" nodes, i.e., nodes without
  any outedges. The dict key is the node the outedge points to and the dict
  value is the weight of that outedge. By default, dangling nodes are given
  outedges according to the personalization vector (uniform if not
  specified). This must be selected to result in an irreducible transition
  matrix (see notes under google_matrix). It may be common to have the
  dangling dict to be the same as the personalization dict.

  Returns
  -------
  pagerank : dictionary
  Dictionary of nodes with PageRank as value
```

```
Notes
-----
The eigenvector calculation is done by the power iteration method
and has no guarantee of convergence. The iteration will stop
after max_iter iterations or an error tolerance of
number_of_nodes(G)*tol has been reached.

The PageRank algorithm was designed for directed graphs but this
algorithm does not check if the input graph is directed and will
execute on undirected graphs by converting each edge in the
directed graph to two edges.


"""
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 for every node. '
              'Missing nodes %s' % missing)
  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 for every node. '
                'Missing nodes %s' % missing)
        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 iterations.' % max_iter)
```

```
import networkx as nx
G=nx.barabasi_albert_graph(60,41)
pr=nx.pagerank(G,0.4)
pr
```

```
{0: 0.028174522007166025,
 1: 0.012964375841888025,
 2: 0.012365727433352219,
 3: 0.01297831400635665,
 4: 0.012965025025177303,
 5: 0.012967949359627928,
 6: 0.013376105183762643,
 7: 0.013375098717208557,
 8: 0.013155256599641786,
 9: 0.01337352326160753,
 10: 0.012781467214759848,
 11: 0.01357008359088016,
 12: 0.012954075308228293,
 13: 0.012345810370752771,
 14: 0.012355176527771825,
 15: 0.012968697150023929,
 16: 0.013358555889755614,
 17: 0.012963636126170228,
 18: 0.013570154828103224,
 19: 0.01236497347992989,
 20: 0.013170002404273538,
 21: 0.013358555889755614,
 22: 0.013172287867239079,
 23: 0.012183840489163334,
```

```
24: 0.013357076706716818,
25: 0.013366202608781182,
26: 0.012360566665586973,
27: 0.013359607456725297,
28: 0.013182744673484674,
29: 0.012343228448597659,
30: 0.013362268690781678,
31: 0.012578854138999066,
32: 0.01215499487946723,
33: 0.013169645722994077,
34: 0.013378184993187365,
35: 0.013154881406752309,
36: 0.012163467197554205,
37: 0.013567422356823778,
38: 0.012383063305193284,
39: 0.013377707431427124,
40: 0.011770835439405396,
41: 0.012961128433131242,
42: 0.027551617097250805,
43: 0.0272384921802679,
44: 0.027124368130833174,
45: 0.026361572844632773,
46: 0.02608214715189193,
47: 0.02569277875560696,
48: 0.025347148413461838,
49: 0.025331997661989403,
50: 0.024596139246889248,
51: 0.024410430624380468,
52: 0.024104557067671444,
53: 0.02342382294839428,
54: 0.023208664577523706,
55: 0.022912776534078798,
56: 0.02239579777701904,
```