**Nama : Rizki Cesta Widiyani**

**NIM : 15.01.53.0139**

**TUGAS ANALISA JEJARING SOSIAL**

# 1. Graph Betweenness Centrality

Graph graph **=** **new** SingleGraph**(**"Betweenness Test"**);**

*// E----D AB=1, BC=5, CD=3, DE=2, BE=6, EA=4*

*// /| | Cb(A)=4*

*// / | | Cb(B)=2*

*// A | | Cb(C)=0*

*// \ | | Cb(D)=2*

*// \| | Cb(E)=4*

*// B----C*

Node A **=** graph**.**addNode**(**"A"**);**

Node B **=** graph**.**addNode**(**"B"**);**

Node E **=** graph**.**addNode**(**"E"**);**

Node C **=** graph**.**addNode**(**"C"**);**

Node D **=** graph**.**addNode**(**"D"**);**

graph**.**addEdge**(**"AB"**,** "A"**,** "B"**);**

graph**.**addEdge**(**"BE"**,** "B"**,** "E"**);**

graph**.**addEdge**(**"BC"**,** "B"**,** "C"**);**

graph**.**addEdge**(**"ED"**,** "E"**,** "D"**);**

graph**.**addEdge**(**"CD"**,** "C"**,** "D"**);**

graph**.**addEdge**(**"AE"**,** "A"**,** "E"**);**

BetweennessCentrality bcb **=** **new** BetweennessCentrality**();**

bcb**.**setWeightAttributeName**(**"weight"**);**

bcb**.**setWeight**(**A**,** B**,** 1**);**

bcb**.**setWeight**(**B**,** E**,** 6**);**

bcb**.**setWeight**(**B**,** C**,** 5**);**

bcb**.**setWeight**(**E**,** D**,** 2**);**

bcb**.**setWeight**(**C**,** D**,** 3**);**

bcb**.**setWeight**(**A**,** E**,** 4**);**

bcb**.**init**(**graph**);**

bcb**.**compute**();**

System**.**out**.**println**(**"A="**+** A**.**getAttribute**(**"Cb"**));**

System**.**out**.**println**(**"B="**+** B**.**getAttribute**(**"Cb"**));**

System**.**out**.**println**(**"C="**+** C**.**getAttribute**(**"Cb"**));**

System**.**out**.**println**(**"D="**+** D**.**getAttribute**(**"Cb"**));**

System**.**out**.**println**(**"E="**+** E**.**getAttribute**(**"Cb"**));**

2.Graph Basic Page Rank

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)