NetworkX: Notes

**NetworkX**

[**Introduction**](#_kkqnzla2ozxn) **4**

[Links](#_a8gnr6pn6kz3) 4

[Installation](#_ncopcmvdnb6x) 5

[Example: Shortest Path](#_gbn1x8zco08g) 5

[Reading the Table of Contents](#_4zo1r3izklmw) 6

[**(Algo) Centrality Measures**](#_bigfpkjsylhu) **7**

[**Closeness Centrality**](#_p8xkdid3wpf9) **7**

[Overview](#_vpxie2d1krsf) 7

[Functions (Shortest Path)](#_3bkeo33fz5j0) 8

[closeness\_centrality(G, u=None, distance=None, wf\_improved=True)](#_jt2g3rxuly0e) 8

[group\_closeness\_centrality(G, S, weight=None)](#_53faswmt6qla) 9

[incremental\_closeness\_centrality(G, edge, prev\_cc=None, insertion=True, wf\_improved=True)](#_rw2x2vp28up9) 10

[Purpose](#_75fxqk7nebux) 10

[**Betweenness Centrality**](#_mh1122nhbct0) **11**

[Overview](#_ytiops2d3p29) 11

[Functions (Shortest Path)](#_ou5z8k2hipfs) 12

[betweenness\_centrality(G, k=None, normalized=True, weight=None, endpoints=False, seed=None)](#_if97ozqhubuo) 12

[Functions (Current Flow)](#_xa5xhzdwgfs5) 13

[Purpose](#_sb0hf17geawi) 13

[**Degree Centrality**](#_6qbjluxaxdbe) **14**

[Overview](#_xy5mlefnq6og) 14

[Functions](#_fen8wcbtw8jk) 14

[degree\_centrality(G)](#_4ja9dhcucw9n) 14

[The degree centrality for a node v is the fraction of nodes it is connected to.](#_d1jcpf5gnqz3) 15

[Parameters:](#_q330bi319yqm) 15

[G (graph) – A networkx graph](#_q330bi319yqm) 15

[Returns:](#_q330bi319yqm) 15

[nodes – Dictionary of nodes with degree centrality as the value.](#_q330bi319yqm) 15

[Return type](#_q330bi319yqm) 15

[dictionary](#_zeptdnqp5icm) 15

[in\_degree\_centrality(G)](#_voyofmkbxp6j) 15

[out\_degree\_centrality(G)](#_pta78xqxnl7b) 15

[Purpose:](#_xvhzuloedg7e) 16

[**Eigenvector Centrality**](#_dp12w2bmnix5) **17**

[Overview](#_ed2tse3nsusq) 17

[Functions](#_1ax8wzk4yqay) 19

[eigenvector\_centrality(G, max\_iter=100, tol=1e-06, nstart=None, weight=None)](#_opd263vgr8qu) 19

[**Katz Centrality**](#_em3n4su40mgu) **20**

[Overview](#_xfblalhv7s8v) 20

[**Subgraph**](#_i9gntv131fjx) **21**

[Overview](#_jz2sncphuyk) 21

[Functions:](#_9dxss256acwn) 21

[subgraph\_centrality(G)](#_tbo770fs34pb) 21

[subgraph\_centrality\_exp(G)](#_u2kqt4qucueh) 22

[estrada\_index(G)](#_4u617rbcjlv1) 23

[Purpose:](#_48llsz4nx0a) 23

[**(Algo) Link Analysis**](#_zcyyvb4qdg7r) **24**

[**HITS Algorithm - Hubs and Authorities**](#_ldjz9p6sg3h8) **25**

[Overview](#_dm5msygmiji5) 25

[Functions](#_2bkkehh3wmkp) 26

[hits(G, max\_iter=100, tol=1e-08, nstart=None, normalized=True)](#_7uof46ofmtvw) 26

[hits\_numpy(G, normalized=True)](#_gz72matdsyqz) 26

[Authoritative Sources in a Hyperlinked Environment, Jon M. Kleinberg](#_x6g7g9afrt9n) 27

[**PageRank**](#_b7mudssnufmp) **28**

[Overview](#_oaopsqcncm8m) 28

[**(Algo) Similarity Measures**](#_ehz9yvv0h7bc) **30**

[Graph Similarity](#_461h4q7xvxv8) 32

[Graph Edit Distance](#_wtcmmn856t5b) 32

[Edit Distance](#_teicm9j0dcdc) 33

[Statistical Methods](#_o6edwmkcdg) 36

[Iterative Methods](#_rgm2okqszrko) 36

[Existing Functions](#_b3qqo0gnc427) 37

[Simrank Similarity](#_sqrp2mqjwuhd) 37

[Graph Edit Distance (GED)](#_74euol3j54e8) 37

[Potential/Proposed Functions](#_a4y457hozbzt) 38

[Similarity Flooding](#_9x01fjydq3zm) 38

[Scoring Method for Phylogenetic Tree Construction](#_zhb6rvkdgl0e) 38

[Vertex Similarity Method](#_2l6kqggl05ja) 38

[Scoring and Matching Algorithm](#_2xwb9zidnlyo) 38

[Belief Propagation](#_mcxjun98l3xi) 39

# Introduction

[Python’s NetworkX](https://networkx.github.io/documentation/stable/index.html) ([Hagberg et al. 2008](http://conference.scipy.org/proceedings/SciPy2008/paper_2/)).

NetworkX is a Python library that can be used to study graphs and networks. Taken from Wikipedia,

some features include:

* [Classes](https://en.wikipedia.org/wiki/Class_(computer_programming)) for graphs and [digraphs](https://en.wikipedia.org/wiki/Directed_graph).
* Conversion of graphs to and from several formats.
* Ability to construct [random graphs](https://en.wikipedia.org/wiki/Random_graph) or construct them incrementally.
* Ability to find [subgraphs](https://en.wikipedia.org/wiki/Glossary_of_graph_theory#subgraph), [cliques](https://en.wikipedia.org/wiki/Clique_(graph_theory)), [k-cores](https://en.wikipedia.org/wiki/Degeneracy_(graph_theory)).
* Explore [adjacency](https://en.wikipedia.org/wiki/Glossary_of_graph_theory_terms#adjacent), [degree](https://en.wikipedia.org/wiki/Degree_(graph_theory)), [diameter](https://en.wikipedia.org/wiki/Glossary_of_graph_theory#diameter), [radius](https://en.wikipedia.org/wiki/Glossary_of_graph_theory#radius), [center](https://en.wikipedia.org/wiki/Glossary_of_graph_theory#center), [betweenness](https://en.wikipedia.org/wiki/Betweenness), etc.
* Draw networks in 2D and 3D.

This library provides a vast number of algorithms for determining centrality and eigenvalues, which can be used to pinpoint certain prominent entities within and across groups, including the ‘leaders’, ‘hubs’ and ‘authorities’ evident in the corpus (see [Kleinberg 1997](https://www.cs.cornell.edu/home/kleinber/auth.pdf); Freeman 1977).

From the developers of NetworkX:

*“NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.”*

> Software for complex networks

* Data structures for graphs, digraphs, and multigraphs
* Many standard graph algorithms
* Network structure and analysis measures
* Generators for classic graphs, random graphs, and synthetic networks
* Nodes can be "anything" (e.g., text, images, XML records)
* Edges can hold arbitrary data (e.g., weights, time-series)
* Open source [3-clause BSD license](https://raw.githubusercontent.com/networkx/networkx/master/LICENSE.txt)
* Well tested with over 90% code coverage
* Additional benefits from Python include fast prototyping, easy to teach, and multi-platform

It is also being developed (partially) at Berkeley. You can read more about it at <https://bids.berkeley.edu/research/networkx>.

## Links

Documentation/Reference: <https://networkx.org/documentation/stable/reference/index.html>

Website: <https://networkx.github.io>

Github: <https://github.com/networkx/networkx>

Bug reports: <https://github.com/networkx/networkx/issues>

## Installation

Install the latest version of NetworkX:

$ pip install networkx

Install with all optional dependencies:

$ pip install networkx[all]

## Example: Shortest Path[[1]](#footnote-0)

***Problem***: Find the shortest path between two nodes in an undirected graph.

***Setup:***

import networkx as nx

G = nx.Graph()

G.add\_edge('A', 'B', weight=4)

G.add\_edge('B', 'D', weight=2)

G.add\_edge('A', 'C', weight=3)

G.add\_edge('C', 'D', weight=4)

***Solution***:

# Finding the shortest path using NetworkX

nx.shortest\_path(G, 'A', 'D', weight='weight')

***Output***:

['A', 'B', 'D']

## Reading the Table of Contents

Sections beyond here will be labeled by the following categories:

* (Algo) : Algorithms

# (Algo) Centrality Measures

## Closeness Centrality

### Overview

Closeness centrality is a metric used to describe how “close” a node is to all other nodes in a network. More specifically, it measures the average distance to all other nodes (and takes the inverse). Therefore, nodes with a high closeness score have a relatively shorter distance to all other nodes. By this definition, a central node is expected to have a higher closeness centrality than nodes that are less central.

It is defined by the formula:



It may be simpler to first consider the reciprocal of this formula,

.

The top part represents the sum of the distances of node u to every other node, by finding the shortest distance between u and every other node, represented by v. The bottom part divides by the number of nodes outside of u, so the equation as a whole represents the average distance from u to any outside node in the network. Therefore, having a higher value means having a greater average distance. The closeness centrality is found by taking the inverse of this formula, so that a higher value means having a shorter average distance from other nodes, or being “close” to other nodes.

Example:

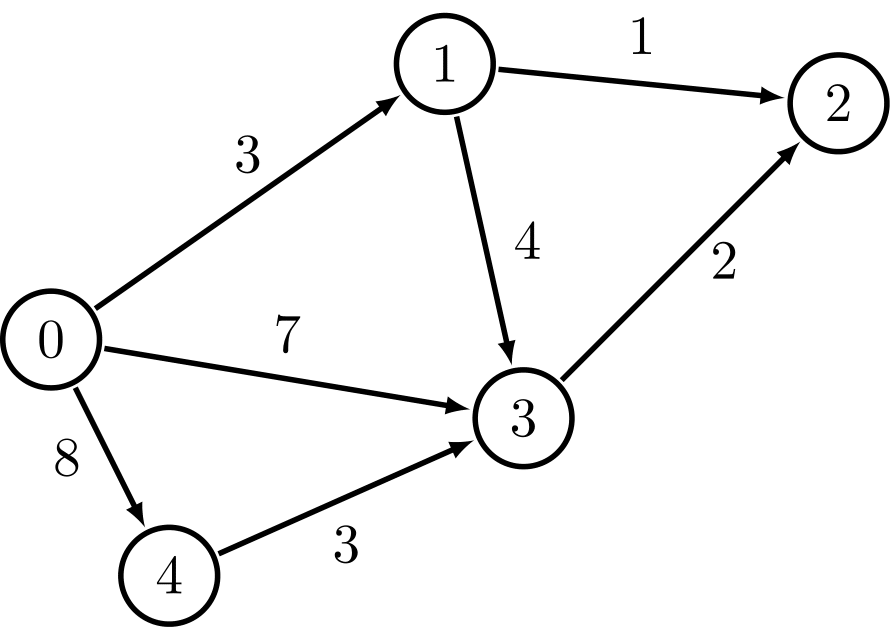


Figure A1

Let u = node 3.

* The shortest distances from u to every other node are 7, 4, 2 and 3, going in numerical order, so = 7 + 4 + 2 + 3 = 16.
* = 4, since there are 5 nodes in total, or 4 nodes other than the node u itself.

C(u) == 4 / 16 = 1 / 4.

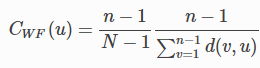
Notice that if node 3 is pulled far away from the rest of the nodes, then would become significantly larger, while the would not change, thus decreasing the “closeness” of u to other nodes.

### Functions (Shortest Path)

#### closeness\_centrality(G, u=None, distance=None, wf\_improved=True)

Explanation:

closeness\_centrality uses an “improved” formula that has slight balancing effects on the calculation:



where the C(u) is multiplied by (n-1)/(N-1). n represents the number of nodes that can reach u (including u), and N represents the total number of nodes in the graph.

Returns: Closeness centrality, C(u)

Given:

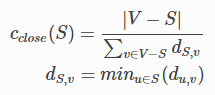
* G (graph)
* u (node)
  + If left blank, then the closeness centrality is calculated for all nodes.
* distance (edge attribute key)
  + Most likely left blank. If given, then Djikstra’s algorithm will be used with the given edge attribute.
* wf\_improved (bool)
  + If left blank, the default uses the “improved” formula. Set to false to use the classical closeness centrality formula as described in the overview.

Note: The average distance can be found instead by passing in G.reverse() into the argument G.

#### group\_closeness\_centrality(G, S, weight=None)

Explanation:

Calculates the closeness centrality for a group of nodes:



where S represents all the nodes for a particular group.

The first equation is very similar to the closeness centrality formula. |V - S| represents the number of nodes that are not in the set S, just like how was used in the numerator of the closeness centrality formula to denote all nodes other than the singleton set containing u. The denominator of the first equation, combined with the second equation, considers every node that is not in the set S and finds its shortest distance to any node in S. For example, going back to [Figure A1](#ipp1mxfe25ti), if we assume nodes 1, 2 and 3 to be part of S, then nodes 0 and 4 are in |V - S|. For node 0, its shortest distance to nodes 1, 2 and 3 are 3, 4 and 7, respectively, so dS, v would be the minimum of these values, or 3. Doing the same for node 4, we get a value of 3 as well and so summing up , 3 + 3 = 6. Since |V - S| = 2, then the centrality would be 2/6 = 1/3.

Other notes / edge cases can be found [here](https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.group_closeness_centrality.html#networkx.algorithms.centrality.group_closeness_centrality).

Returns: Closeness centrality for a group, C(u)

Given:

* G (graph)
* S (list or set) - nodes
* weight (string)
  + If the graph is unweighted, then this can be left blank.
  + If the graph is weighted, then the weight is used as an [edge attribute](https://networkx.org/documentation/stable/tutorial.html#edge-attributes).

Note: The direction of edges in a weighted graph can be reversed by passing in G.reverse() into the argument G.

#### incremental\_closeness\_centrality(G, edge, prev\_cc=None, insertion=True, wf\_improved=True)

### Purpose

In AWCA, we use closeness centrality to determine the position of each document in a cluster or group of related documents. This algorithm plots a document’s closeness to the center of a group based on the clustering coefficient, which is a product of the document’s degree (i.e. number of edges or links between documents).

## 

## Betweenness Centrality

### Overview

Betweenness centrality is a metric used to score how much a node is “in-between” all other nodes. Rather than relying more heavily on the weights or the counts of edges as in closeness centrality, betweenness centrality emphasizes the structure and connectedness of a given network with regards to a node u. More specifically, it captures a node’s ability to control the flow of communication, by counting how many times a node is found on the shortest path between two other nodes. With a higher betweenness centrality score, a node has more power to control communication since much of the information being transmitted will flow through the node.

It is defined by the formula:



where v represents the node we are trying to score, and V represents the set of all nodes in the network.

The function finds the number of shortest paths between any two nodes. So for any two nodes in the network which will be chosen as s and t, is the number of shortest paths between s and t, while is the number of shortest paths between s and t that include v somewhere in between. Therefore, for any two nodes s and t, the fraction in the equation is equivalent to how often v is contained in a shortest path between any two nodes. The equation then considers all possible combinations of two nodes, sums up the value of / ) for each combination, and returns that as the betweenness centrality.

When s = v or t = v, then = 0. When s = t, then = 1.

Example:

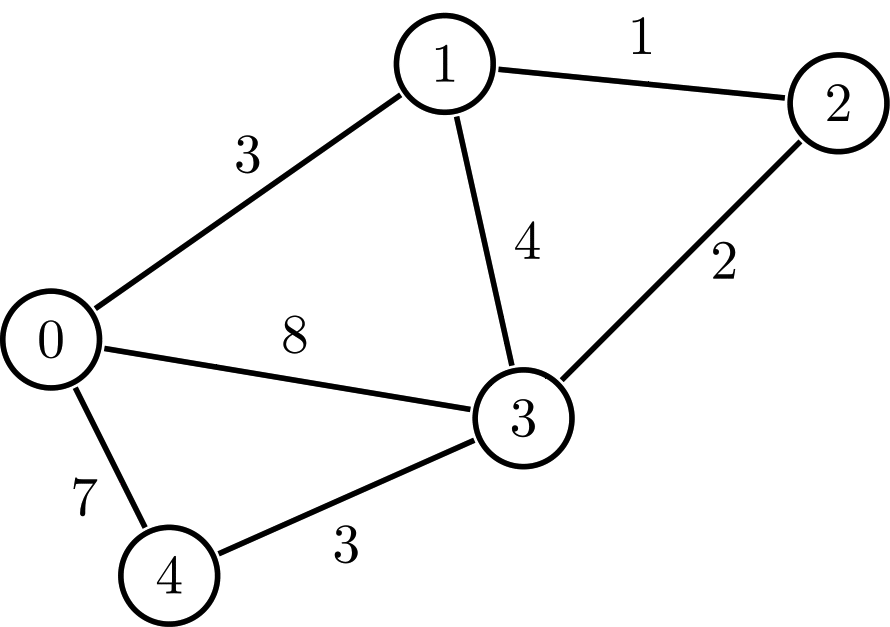


Figure A2

Let v = node 1.

For node 0, there is one shortest path to each node,

* 0-0 (1),
* 0-1 (0),
* 0-1-2 (1),
* 0-1-2-3 (1), (which is the shortest path because 0-3 has distance 8 to 0-1-2-3’s 6)
* 0-4 (0).

Repeating this for other nodes, we see that node 1 scores 0 points (since = 0), node 2 scores 1 point (excluding 0-2, which was already considered), node 3 scores 1 point (3-3), and node 4 scores 1 point (4-4). So node 1 has a betweenness centrality of 3+1+1+1 = 6.

The betweenness centrality can seemingly become infinite as the number of nodes increases, so the betweenness centrality is only useful by comparing it to the centrality of other nodes in the network.

### Functions (Shortest Path)

#### betweenness\_centrality(G, k=None, normalized=True, weight=None, endpoints=False, seed=None)

Explanation:

Calculates betweenness centrality for all nodes:



as described in the overview.

Returns: Betweenness a dictionary with the nodes of the network as the keys and the betweenness centrality as the values.

Given:

* G (graph)
* k (int)
  + Only use if runtime becomes an issue. Limits the number of node pairs considered to decrease runtime while sacrificing accuracy.
* normalized (bool)
* weight (string)
* endpoints (bool)
  + If set to true, then = 1. The scores will be higher for all nodes, which will slightly change the distribution of centralities among nodes. Default is false.
* seed (int or random\_state)
  + Only used if k is not None, because otherwise there’s no randomness as all node pairs are considered.

### Functions (Current Flow)

### Purpose

## 

## Degree Centrality

### Overview

Definition : **Degree** of vertex of a graph () is the number of edges incident to the vertex .

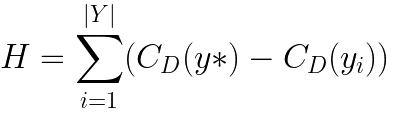
Definition: **Degree centrality** assigns an importance score based on the vertex’s degree.

The **degree centrality** of a vertex , for a given graph with vertices and edges, is defined as:



High degree centrality indicates that the node has a large number of neighbors and is closely related to its neighbors.

To calculate the **degree centrality** of the graph , let be the node with the highest degree centrality in . Let be the node connected graph that maximizes the following quantity:



Then, the **degree centrality** of the graph is:



Example: Degree centrality would target the most popular one in the network ⇒ e.g. Most popular topic.

Reference: <https://en.wikipedia.org/wiki/Centrality#Degree_centrality>

### Functions

#### 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 of nodes with degree centrality as the value.

### Return type

### dictionary

#### in\_degree\_centrality(G)

Compute the in-degree centrality for nodes.

The in-degree centrality for a node v is the fraction of nodes its incoming edges are connected to.

Parameters:

G (graph) – A NetworkX graph

Returns:

nodes – Dictionary of nodes with in-degree centrality as values.

Return type:

dictionary

Raises:

[NetworkXNotImplemented](https://networkx.org/documentation/stable/reference/exceptions.html#networkx.NetworkXNotImplemented) – If G is undirected.

#### out\_degree\_centrality(G)

Compute the out-degree centrality for nodes.

The out-degree centrality for a node v is the fraction of nodes its outgoing edges are connected to.

Parameters:

G (graph) – A NetworkX graph

Returns:

nodes – Dictionary of nodes with out-degree centrality as values.

Return type:

dictionary

Raises:

[NetworkXNotImplemented](https://networkx.org/documentation/stable/reference/exceptions.html#networkx.NetworkXNotImplemented) – If G is undirected.

### Purpose:

## Eigenvector Centrality

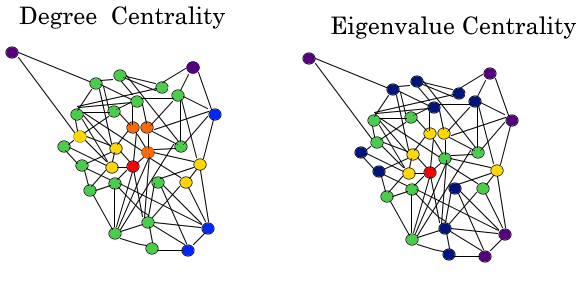
### Overview

Eigenvector centrality, also known as eigencentrality, is another metric used to describe the influence a node has on a network. In the previous section, we saw that a node’s degree centrality could be calculated simply by counting the number of edges it has, thus making it one of the simplest centrality metrics to calculate. Eigenvector centrality can be thought of as an extension to degree centrality, where in addition to considering the number of edges a node has, we must also be concerned with the importance or centrality of the nodes to which the current node, u, connects to. As stated by the eigenvector centrality thesis:

*A node is important if it is linked to by other important nodes*[[2]](#footnote-1).

Another way to put this is to think of eigenvector centrality as a weighted form of degree centrality. In degree centrality, you can treat every node as a node of weight 1, and to find the centrality, you can take the sum of the node weights that are adjacent to node u. In eigenvector centrality, you must first figure out how to assign weights to each node, at the same time you are trying to calculate a node’s centrality a node’s weight; thus, the calculation is at least a bit more complicated than degree centrality, although that would be putting it quite lightly.

Below is a comparison of degree centrality against eigenvector centrality. As seen from the graphs, degree centrality has a harder time figuring out which nodes are the most important, as it assigns the same centrality score to more than half of its nodes. Eigenvalue centrality, on the other hand, eliminates many of these green nodes near the edges that it deems to be less relevant on the network than other green nodes. However, eigenvector centrality sometimes turns out to be too sensitive.



Each node’s centrality value is calculated using every other node’s final centrality value. This sort of dependency means a solution through elimination is not possible, so the nodes cannot be calculated in sequential order, unlike the types of centrality we have already observed. This gives rise to the use of eigenvectors and eigenvalues, a key component of linear algebra with applications in all kinds of subjects.

When used in graph theory, the [adjacency matrix](https://mathworld.wolfram.com/AdjacencyMatrix.html) is particularly useful in the calculation of the centrality value. The number of rows and columns are both equal to the number of nodes, one for each node, where the edge from a value from the adjacency matrix aij is 1 if an edge exists from i to j, and 0 otherwise. We use

Ax = λx,

where x, the centrality, is represented by



or



where x′=Ax.

For some node v, the centrality of node v is determined by summing over the final centralities of all other nodes, where the other nodes are represented by t (scaled by λ).

In NetworkX, the adjacency matrix, A, is determined by passing in a graph G, and the rest can be found with the use of eigenvectors. Thus, passing in the graph alone is enough to find the eigenvector centrality.

As a final note, Google’s PageRank and the Katz centrality are both based on eigenvector centrality. Standard eigenvector centrality tends to be flawed in that it only works with strongly connected networks, which directed networks often do not have. PageRank relies on backlinks and in-degrees to be able to be used in directed networks, which allows it to be so effective in a social network such as Google search. The Katz centrality measure, which is a part of the NetworkX library, makes some changes to the eigenvector centrality to work well on acyclic directed networks. An alpha value must be chosen and tweaked, as low alpha values tend to cause the Katz score to be determined by short paths, while large values take longer paths into account. For more information on why this works and how alpha should be chosen, a simple description is [here](https://www.sci.unich.it/~francesc/teaching/network/katz.html). An example can also be found [here](https://www.geeksforgeeks.org/katz-centrality-centrality-measure/). These measures will be elaborated on later.

### Functions

#### eigenvector\_centrality(G, max\_iter=100, tol=1e-06, nstart=None, weight=None)

## 

## Katz Centrality

### Overview

Katz centrality introduces two positive constants α and β:





where Aij is an element of the adjacency matrix.

* The β constant gives a free centrality contribution for all nodes even though they don’t get any contribution from other nodes. The existence of a node alone would provide it some importance.
* The α constant determines the balances between the contribution from other nodes and the free constant.

Although this method is introduced as a solution for directed graphs, it can be useful for some applications of undirected graphs as well.

## Subgraph

### Overview

Definition: **Subgraph centrality** accounts for the participation of a node in all sub graphs of the network:

Subgraph Centrality

**Subgraph centrality** of a node n is the sum of weighted closed walks of all lengths starting and ending at node n. The weights decrease with path length. Each closed walk is associated with a connected subgraph ([1](https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.subgraph_centrality.html#id4)). It is just the diagonal entry of the exponential of the adjacency matrix.

Definition**:** Let be a simple undirected graph with n nodes and let λ1 ≤ λ2 ≤ ⋯ λn be a non-increasing ordering of the eigenvalues of its adjacency matrix A. The **Estrada index** is:



[[A HAGBERG, D. S., 2008, ESTRADA, E. 2000]](https://doi.org/10.1016/S0009-2614(00)00158-5)

### Functions:

#### subgraph\_centrality(G)

Returns subgraph centrality for each node in G.

Parameters:

G (graph)

Returns:

nodes – Dictionary of nodes with subgraph centrality as the value.

Return type:

dictionary

Raises:

NetworkXError – If the graph is not undirected and simple.

**Notes:**

This version of the algorithm computes **eigenvalues and eigenvectors of the adjacency matrix**. 

Subgraph centrality of a node u in G can be found using a spectral decomposition of the adjacency matrix ([1](https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.subgraph_centrality.html#id4)), where v\_j is an eigenvector of the adjacency matrix A of G corresponding corresponding to the eigenvalue lambda\_j.

**Examples:**

(Example from [1](https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.subgraph_centrality.html#id4))

>>> G = nx.Graph( … [ … (1, 2), … (1, 5), … (1, 8), … (2, 3), … (2, 8), … (3, 4), … (3, 6), … (4, 5), … (4, 7), … (5, 6), … (6, 7), … (7, 8), … ] … )

>>> sc = nx.subgraph\_centrality(G)

>>> print([f”{node} {sc[node]:0.2f}” for node in sorted(sc)])

>>> [‘1 3.90’, ‘2 3.90’, ‘3 3.64’, ‘4 3.71’, ‘5 3.64’, ‘6 3.71’, ‘7 3.64’, ‘8 3.90’]

#### subgraph\_centrality\_exp(G)

Returns the subgraph centrality for each node of G. Subgraph centrality of a node n is the sum of weighted closed walks of all lengths starting and ending at node n. The weights decrease with path length. Each closed walk is associated with a connected subgraph ([1](https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.subgraph_centrality.html#id4)).

Parameters:

G (graph)

Returns:

nodes – Dictionary of nodes with subgraph centrality as the value.

Return type:

dictionary

Raises:

NetworkXError – If the graph is not undirected and simple.

**Notes:**

This version of the algorithm exponentiates the adjacency matrix. The subgraph centrality of a node u in G can be found using the matrix exponential of the adjacency matrix of G ([1](https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.subgraph_centrality.html#id4)):



**Examples**([1](https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.subgraph_centrality.html#id4)):

(Example from [1](https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.subgraph_centrality.html#id4))

>>> G = nx.Graph( … [ … (1, 2), … (1, 5), … (1, 8), … (2, 3), … (2, 8), … (3, 4), … (3, 6), … (4, 5), … (4, 7), … (5, 6), … (6, 7), … (7, 8), … ] … )

>>> sc = nx.subgraph\_centrality\_exp(G)

>>> print([f”{node} {sc[node]:0.2f}” for node in sorted(sc)])

>>> [‘1 3.90’, ‘2 3.90’, ‘3 3.64’, ‘4 3.71’, ‘5 3.64’, ‘6 3.71’, ‘7 3.64’, ‘8 3.90’]

#### estrada\_index(G)

Returns the Estrada index of a graph G.

The Estrada Index is a topological index of folding or 3D “compactness”([1](https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.centrality.subgraph_centrality.html#id4)).

Parameters:

G (graph)

Returns:

estrada index (float): See the definition of estrada\_index in [Overview](#_jz2sncphuyk)

Return type:

float

Raises:

NetworkXError – If the graph is not undirected and simple.

### Purpose:

# 

# (Algo) Link Analysis

<https://www.csc2.ncsu.edu/faculty/nfsamato/practical-graph-mining-with-R/sample/chapter_5_LinkAnalysis.pdf> - 5.4, 5.5, 5.7 (optional: 5.1, 5.2) (use for intro)

## HITS Algorithm - Hubs and Authorities

### Overview

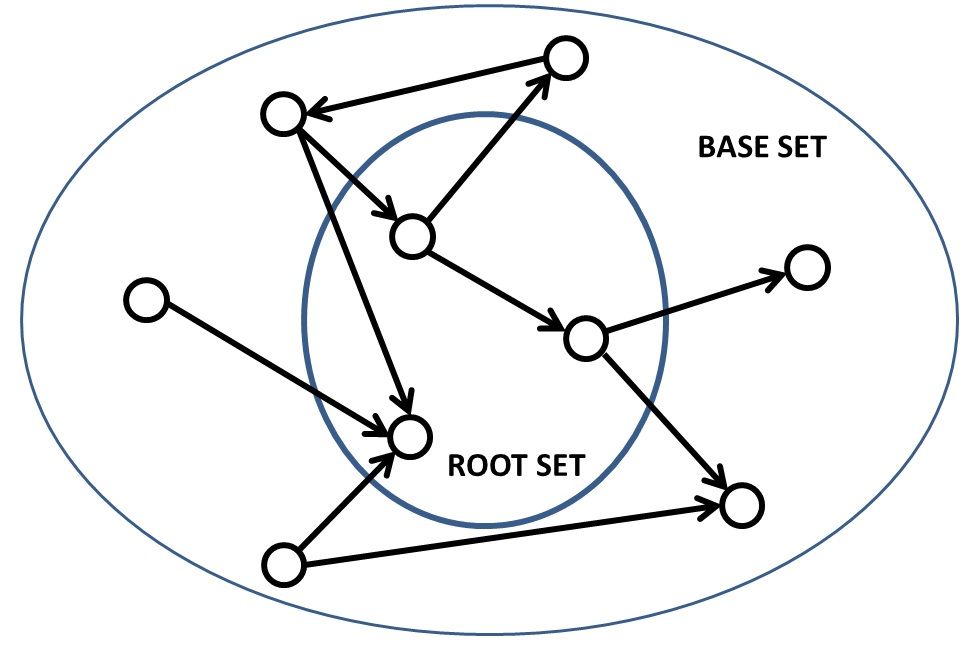
All of the other measures capture high node centrality; however, there can also be nodes that are useful and yet not central. For example, a review article in a scientific citation network would not be finding any new discoveries and thus would not be central, yet it could still be a useful reference in trying to acquire knowledge on a topic because it captures a lot of central research articles. In the HITS algorithm, two types of its own central nodes: *Hubs* and *Authorities*.

Definition: We begin by retrieving the most relevant pages to the search query. This set is called the **root set**.

Definition: A **base set** is generated by augmenting the root set with all the web pages that are linked from it and some of the pages that link to it. The purpose of tracking base set is to ensure that most (or many) of the strongest authorities are included.

Definition: **Authorities** are the one that most cited by Hubs and Hubs are the one that citing the most high Authority nodes. An **authority value** is computed as the sum of the scaled **hub values** that point to that page. A **hub value** is the sum of the scaled authority values of the pages it points to.

The two basic steps of Hits are:



* **Authority update**: Update each node's authority score to be equal to the sum of the hub scores of each node that points to it.
* **Hub update**: Update each node's hub score to be equal to the sum of the authority scores of each node that it points to.

Reference: [Wikipedia](https://en.wikipedia.org/wiki/HITS_algorithm)

Authority centrality



Hub centrality





### Functions

#### hits(G, max\_iter=100, tol=1e-08, nstart=None, normalized=True)

Parameters:

G (graph) – A NetworkX graph

max\_iter (integer, optional) – Maximum number of iterations in power method.

tol (float, optional) – Error tolerance used to check convergence in power method iteration.

nstart (dictionary, optional) – Starting value of each node for power method iteration.

normalized (bool (default=True)) – Normalize results by the sum of all of the values.

Returns:

(hubs,authorities) – Two dictionaries keyed by node containing the hub and authority values.

Return type:

two-tuple of dictionaries

Raises:

PowerIterationFailedConvergence – If the algorithm fails to converge to the specified tolerance within the specified number of iterations of the power iteration method.

#### hits\_numpy(G, normalized=True)

Returns HITS hubs and authorities values for nodes.

Parameters:

G (graph) – A NetworkX graph

normalized (bool (default=True)) – Normalize results by the sum of all of the values. Returns:

(hubs,authorities) – Two dictionaries keyed by node containing the hub and authority values.

Return type:

two-tuple of dictionaries

### Authoritative Sources in a Hyperlinked Environment, Jon M. Kleinberg

## PageRank

### Overview

From Wikipedia:

*"PageRank is an algorithm used by Google Search to rank websites in their search engine results. PageRank was named after Larry Page, one of the founders of Google. PageRank is a way of measuring the importance of website pages. According to Google: PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.”*

There are four common frameworks by which academics view Google's PageRank algorithm:

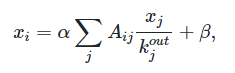
* The first looks at the social impact, both positive and negative, of immediate access to previously unimaginable knowledge through one centralized terminal.
* The second, and most mathematical, sees PageRank as a computation of the Singular Value Decomposition (SVD (http://en.wikipedia.org/wiki/Singular\_value\_decomposition)) of the adjacency matrix of the graph formed by the internet, with particular emphasis paid the the first few singular vectors.
* The third, and most far reaching practical, technical implication of Google's work, is the implementation of algorithms and computation at enormous scale.
* Finally, a more intuitive way to look at the PageRank algorithm is through the lens of a web crawler (or many web crawlers) acting as an agent (or agents) in a Markov Chain the size of the web. More on this last section can be found in *Probability in Electrical Engineering and Computer Science*, by Jean Walrand, in the first two chapters.

As briefly mentioned early, PageRank is built off of eigenvector centrality, but it can also be considered an extension to Katz centrality.

In Katz centrality, we have the formula:

[[3]](#footnote-2)

When considering the purpose and context of PageRank, one key issue with the Katz centrality is when a website has too many links to other websites. If a website with a high Katz centrality has millions of links going to other sites, it would contribute a high value to each site, even though not all of these sites are very useful. For example, a Wikipedia page on the planet Earth may have thousands of links to other Wikipedia sites, and under the Katz centrality each site it links to would be given an extremely high value (due to Earth having a high centrality value itself). Therefore, PageRank modifies the Katz centrality with the following formula:



where kjout represents the out-degree of node j, and

kjout = 1 for zero out-degree nodes (to avoid errors).

This can also be represented in a matrix form,

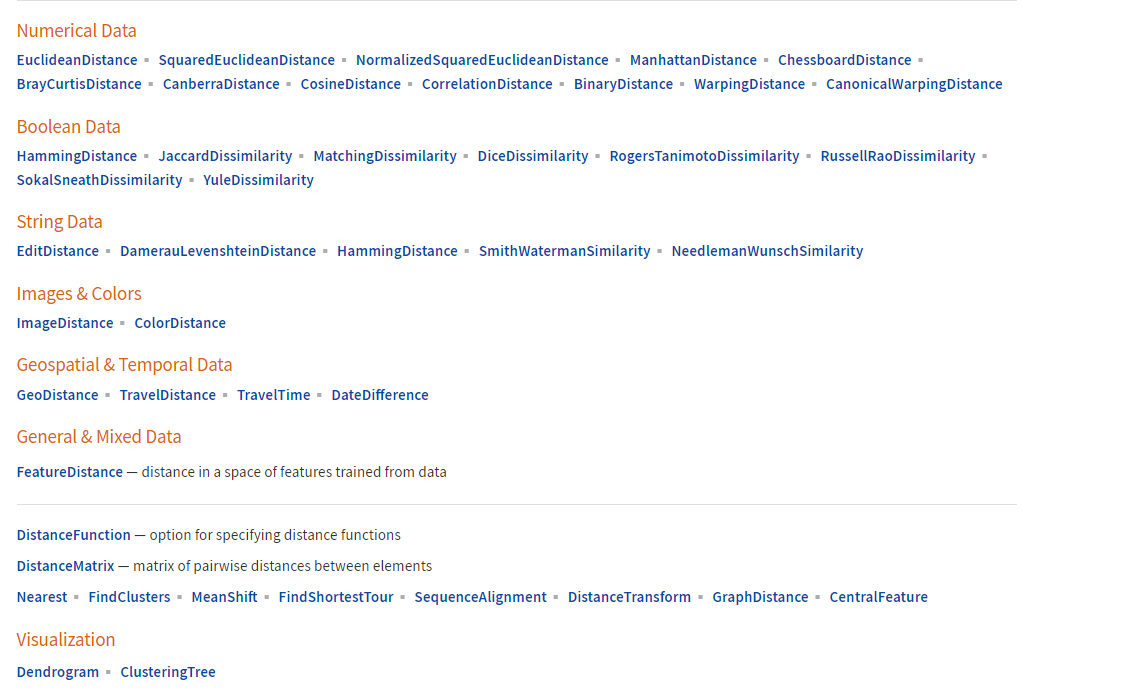


where **D** is a diagonal matrix with elements **Dii** = max(kjout, 1).

# 

# (Algo) Similarity Measures

This is listed under the section “Similarity Measures” in NetworkX, which is a pretty broad topic. In fact, on Wolfram, they have:



In NetworkX, the focus is on graphs, so anything related to graph similarity measures goes in (excluding a large majority of what is shown above). In particular, there is an emphasis being placed on graph edit distance. The current documentation is also quite new and bound for future updates, so this section may be out of date pretty soon. However, rather than focusing on the functions, I’ll try to explain the main concepts / ideas, like usual.

Before going into graph similarity, it might be useful to first have an idea of what types of similarity measures are out there, or what it is in the first place. In general, it is used to measure how alike two data objects are. It’s a very common question for humanists and critics of all kinds: given what you know about two things, how alike or how different are they? Non-computational assessments of similarity and difference form the basis of a lot of critical activity. The genre of a text, for example, can be determined by assessing that text’s similarity to other texts already known to be part of the genre. And conversely, knowing that a certain text is very different from others in an established genre might open up productive new avenues for criticism. An object of study’s uniqueness or sameness relative to another object or to a group can be a crucial factor in the scholarly practices of categorization and critical analysis.

Statistical measures of similarity allow scholars to think computationally about how alike or different their objects of study may be, and these measures are the building blocks of many other clustering and classification techniques[[4]](#footnote-3). The concept of similarity is fundamentally important in almost every scientific field:

* In mathematics, geometric methods for assessing similarity are used in studies of congruence and homothety as well as in allied fields such as trigonometry.
* Topological methods are applied in fields such as semantics
* [Fuzzy set](http://www.scholarpedia.org/article/Fuzzy_Sets) theory has also developed its own measures of similarity, which find application in areas such as management, medicine and meteorology
* In molecular biology, similarity measures are used to measure the sequence similarity of pairs of proteins
* Also in psychological experiments and theories, etc…

And of course….in graph theory! Graph theory is widely used for assessing cladistic similarities in taxonomy[[5]](#footnote-4). In text analysis, the similarity of two texts can be assessed in its most basic form by representing each text as a series of word counts and calculating distance using those word counts as features.

The most common types of similarity measures are numerical in form, including Manhattan distance, Euclidean distance, and cosine similarity. To read more about those and similarity measures in general, you can read here:

* <https://programminghistorian.org/en/lessons/common-similarity-measures> - numerical similarity measures
* <https://dataaspirant.com/five-most-popular-similarity-measures-implementation-in-python/#:~:text=The%20similarity%20measure%20is%20the,a%20high%20degree%20of%20similarity>. - more numerical similarity measures
* <http://www.scholarpedia.org/article/Similarity_measures> - types of similarity measures
* <https://reference.wolframcloud.com/language/guide/DistanceAndSimilarityMeasures.html> - Wolfram’s similaity/distance measures

And, here: <https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf>, for CS170’s textbook on Dynamic Programming, which often efficiently solves distance problems. Section 6.3 covers edit distance, although this will be covered later as well.

## Graph Similarity

In graph similarity, we have two graphs on the same set of N nodes, but with possibly different sets of edges (weighted or unweighted). We assume that we know the correspondence between the nodes of the two graphs (like the people in PC don’t vary across graphs). Graph similarity involves determining the degree of similarity between these two graphs (a number between 0 and 1).

The applications of graph similarity extend to a wide range of fields, such as social networks, image processing, biological networks, chemical compounds, and computer vision. As such, there are many different algorithms to tackle this problem. The proposed techniques can be classified into three main categories: edit distance/graph isomorphism, feature extraction, and iterative methods.

NetworkX currently supports two graph similarity algorithms:

* Graph Edit Distance (GED)
* Simrank Similarity

There are other options that may fit in or replace the current algorithms.

Intuitively, since we know the node correspondences, the same node in both graphs would be similar if its neighbors are similar (and its connectivity, in terms of edge weights, to its neighbors). Again, its neighbors are similar if their neighborhoods are similar, and so on. This intuition guides the possibility of using belief propagation (BP) as a method for measuring graph similarity, precisely because of the nature of the algorithm and its dependence on neighborhood structure.[[6]](#footnote-5)

### 

### Graph Edit Distance

Isomorphic graphs: Two graphs G1 and G2 are said to be isomorphic if

* Their number of components (vertices and edges) are same.
* Their edge connectivity is retained.

Two graphs are similar if one of the following are true:

* they are isomorphic
* one is isomorphic to a subgraph of the other
* they have isomorphic subgraphs

The drawback of graph isomorphism is that finding the exact solution to GED is NP-Hard, meaning it cannot be solved efficiently (in polynomial time), but rather in exponential time. The graph edit distance is a generalization of the graph isomorphism problem, where the target is to transform one graph to the other by doing a number of operations (additions, deletions, substitutions of nodes or edges, and reversions of edges). This method associates each operation with a cost and it attempts to find the sequence of operations that minimizes the cost of matching the two graphs.

#### Edit Distance

A slight divergence from the more specific topic of graph edit distance would be the larger principle of edit distance.

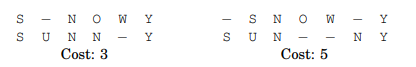
The main idea of (standard) edit distance, or the Levenshtein distance, is to be able to quantify how dissimilar two strings are to one another. Or in other words, it finds the minimum number of edits needed in order to transform one string into another. Here are some examples:

Ex.1

The Levenshtein distance between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

1. kitten → sitten (substitution of "s" for "k")
2. sitten → sittin (substitution of "i" for "e")
3. sittin → sitting (insertion of "g" at the end).

Ex.2



The “−” indicates a “gap”; any number of these can be placed in either string. The cost of an alignment is the number of columns in which the letters differ. And the edit distance between two strings is the cost of their best possible alignment. Do you see that there is no better alignment of SNOWY and SUNNY than the one shown here with a cost of 3?

Edit distance is so named because it can also be thought of as the minimum number of edits—insertions, deletions, and substitutions of characters—needed to transform the first string into the second. For instance, the alignment shown on the left corresponds to three edits: insert U, substitute O → N, and delete W.

Now, to get off topic a bit, distance problems such as edit distance, shortest paths in a graph, and chain matrix multiplication are efficiently solved by dynamic programming, and often a much better method than the brute-force recursive solution. An example of using dynamic programming to solve an edit distance problem is shown below:

Ex. 3

Goal: Find the edit distance between two strings x[1 · · · m] and y[1 · · · n].

Method: When solving a problem by dynamic programming, the most crucial question is, What are the subproblems? Well, it should go part of the way toward solving the whole problem; so how about looking at the edit distance between some *prefix* of the first string, x[1 · · · i], and some *prefix* of the second, y[1 · · · j]? Call this subproblem E(i, j). Our final objective, then, is to compute E(m, n).

For this to work, we need to somehow express E(i, j) in terms of smaller subproblems. Let’s see—what do we know about the best alignment between x[1 · · · i] and y[1 · · · j]? Well, its rightmost column can only be one of three things:



We have no idea which of them is the right one, so we need to try them all and pick the best:

E(i, j) = min{1 + E(i − 1, j), 1 + E(i, j − 1), diff(i, j) + E(i − 1, j − 1)}

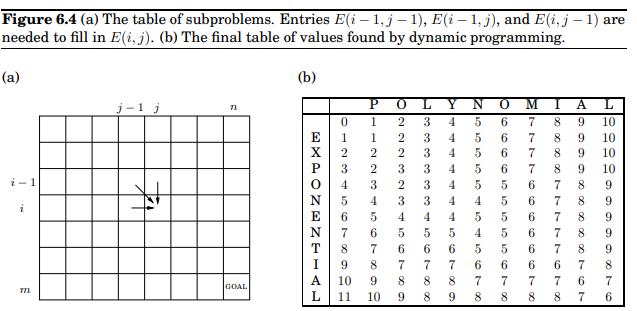
where for convenience diff(i, j) is defined to be 0 if x[i] = y[j] and 1 otherwise.

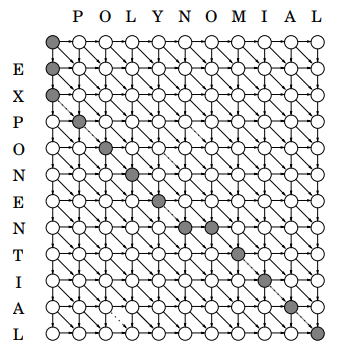
In computing the edit distance between EXPONENTIAL and POLYNOMIAL, subproblem E(4, 3) corresponds to the prefixes EXPO and POL. The rightmost column of their best alignment must be one of the following:



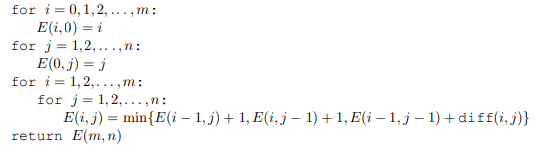
Thus, E(4, 3) = min{1 + E(3, 3), 1 + E(4, 2), 1 + E(3, 2)}.

A good way to visualize this is with a table, where you can move down, right, or both as you start from the top-left corner and must end at the bottom-right.





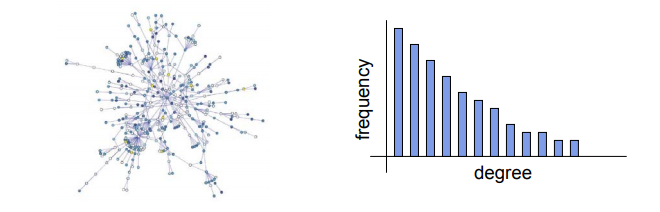
Including the base cases where you are either already at the “bottom wall” or “right wall” and cannot move down or right, respectively, the edit distance algorithm is:



### 

### Statistical Methods

The key idea behind these methods is that similar graphs probably share certain properties, such as degree distribution, diameter, eigenvalues. After extracting these features, a similarity measure is applied in order to assess the similarity between the aggregated statistics and, equivalently, the similarity between the graphs. These methods are powerful and scale well, as they map the graphs to several statistics that are much smaller in size than the graphs. However, depending on the statistics that are chosen, it is possible to get results that are not intuitive. For instance, it is possible to get high similarity between two graphs that have very different node set size, which is not always desirable.

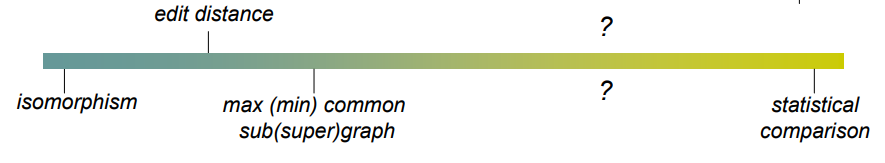
[[7]](#footnote-6)

### Iterative Methods

The philosophy behind the iterative methods is that “two nodes are similar if their neighborhoods are also similar”. In each iteration, the nodes exchange similarity scores and this process ends when convergence is achieved.

Several successful algorithms belong to this category:

* The *similarity flooding algorithm* by Melnik et al., which attempts to find the correspondence between the nodes of two given graphs.
* *SimRank*, which measures the self-similarity of a graph, ie. it assesses the similarities between all pairs of nodes in one graph.
* The *scoring method for phylogenetic tree construction* proposed by Heymans and Singh
* The v*ertex similarity method* of Leicht et al.
* A graph similarity scoring and matching algorithm by Zager and Verghese
* *Belief Propagation*, which proposes two approximate sparse graph matching algorithms using message passing algorithms.



### Existing Functions

Currently, the NetworkX documentation supports HITS, which is covered in a section above, as well as graph edit distance (GED) and SimRank similarity.

#### Simrank Similarity

***SimRank*** - measures the self-similarity of a graph, ie. it assesses the similarities between all pairs of nodes in one graph. It is based on the notion that similar nodes have similar neighborhoods. The algorithm computes iteratively all pairs’ similarity scores, by propagating similarity scores in the A2 matrix, where A is the adjacency matrix of the graph; the process ends when convergence is achieved.

Functions:

simrank\_similarity(G, [source, target])

simrank\_similarity\_numpy(G, [source, target]) - uses numpy

#### Graph Edit Distance (GED)

***Graph Edit Distance*** - defined as minimum cost of edit path (sequence of node and edge edit operations) transforming graph G1 to graph isomorphic to G2.

Functions:

### Potential/Proposed Functions

As pointed out in the NetworkX documentation and in other sources, GED suffers from the issue of being too slow to compute, as it is an NP-hard problem so it runs in exponential time. Statistical methods seem to be covered in other sections, with centrality measures, etc. Therefore, the most potential seems to come out of iterative methods for graph similarity algorithms, SimRank being an example. Some other options to implement:

#### Similarity Flooding

***Similarity Flooding*** - applies in database schema matching; this algorithm solves the “matching” problem, that is, it attempts to find the correspondence between the nodes of two given graphs. What is interesting about the paper is the way the algorithm is evaluated: humans check whether the matchings are correct, and the accuracy of the algorithms is computed based on the number of adaptations that have to be done in the solutions in order to get the right ones.

The goal of similarity flooding is to match elements of related and complex objects, whether this be data schemes, instances, graphs, etc. It is a generic algorithm with wide applicability.

The current approaches include comparing SQL or XMLs of given objects. However, this requires domain-specific knowledge and coding. Similarity flooding relies on structural similarities to find a solution.

At a high level overview, the algorithm is divided into four parts:

* *Input*: finds two objects to match, where the match will be between sub-elements of the two objects.
* *Graph*: represent graphs as directed, labeled graphs.
* *Mapping*: Begin with an initial mapping function σ and an iterative computation of σ, where each σi is based on σi-1 and the current propagation graph. Continue until a stable mapping is reached.
* *Filtering*: Mapping will produce a lot of information, much more than we probably need. Therefore, filtering can remove unnecessary pairs, and the choice of a filter is domain-specific. Some methods include removal by domain-specific information, cardinality, and threshold.

A concise and instructive slideshow can be found [here](https://slidetodoc.com/similarity-flooding-a-versatile-graph-matching-algorithm-by/). The paper is [here](https://www.sciencedirect.com/science/article/pii/S0893965907001012#b7).

#### Scoring Method for Phylogenetic Tree Construction

[scoring method for phylogenetic tree construction proposed by Heymans and Singh](https://www.sciencedirect.com/science/article/pii/S0893965907001012#b9)

***Phylogenetic Tree Construction*** - finds the score between two sequences which provides information about their evolutionary relationship to each other. When more than two sequences are present the scores between all combinations of sequence pairs form the starting point for producing a multiple alignment.

Some of its application in data science:

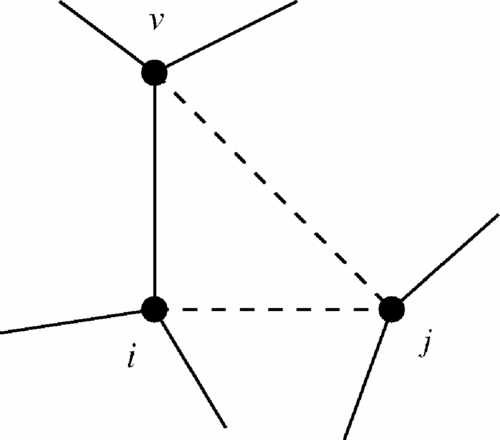
* Ortholog identification
* Ancestral sequence reconstruction
* Synthesized for experimental investigation (e.g., to explore the ancestral function, folding pathways/stability)
* Phylogenomic function prediction
* Exploring the evolution of a functional and structural domain
* Functional site prediction
* Cellular localization and membrane topology prediction
* Modelling the evolution of metabolic and signalling networks and pathways

One of Berkeley’s courses has slides on this with a download [here](https://bcourses.berkeley.edu/files/70012662/download?download_frd=1).

Additional sources: [construction types](http://phylogenetictrees.com/sequences_to_trees.php), [trees](http://phylogenetictrees.com/sequences_to_trees.php), [application based paper](https://www.longdom.org/open-access/computational-analysis-of-distance-and-character-based-phylogenetic-tree-for-capsid-proteins-of-human-herpes-virus-2153-0602.1000128.pdf)

#### Vertex Similarity Method

***Vertex Similarity Method*** - proposes a measure of similarity based on the concept that two vertices are similar if their immediate neighbors in the network are themselves similar.



A vertex j is similar to vertex i (dashed line) if i has a network neighbor v (solid line) that is itself similar to j .

The paper can be found [here](https://pubmed.ncbi.nlm.nih.gov/16605411/).

#### Scoring and Matching Algorithm

***Scoring and Matching Algorithm*** - this method introduces the idea of coupling the similarity scores of nodes and edges in order to compute the similarity between two graphs; the majority of the earlier proposed methods focuses on the nodes’ scores. In this work, the node correspondence is unknown and the proposed algorithm computes the similarity between all pairs of nodes, as well as all pairs of edges, in order to find the mapping between the nodes in the graph.

It presents a related similarity measure that uses a linear update to generate both node and edge similarity scores that do not depend on the initial values of the node scores, then explores the application of this measure to graph matching.

The paper can be found [here](https://core.ac.uk/download/pdf/82715927.pdf).

#### Belief Propagation

Resources : [here](http://people.csail.mit.edu/billf/publications/Understanding_Belief_Propogation.pdf), [wiki](https://en.wikipedia.org/wiki/Belief_propagation), and most useful would be the paper referenced earlier, [here](https://www.cs.cmu.edu/~jingx/docs/DBreport.pdf).

# (Algo) Node Classification

## Harmonic Function

### Overview:

##### Definition: According to Wolfram MathWorld, any real function u(x,y) with continuous second partial derivatives which satisfies is called a **harmonic function**. In this section, the algorithm is developed by Zhu et al. to compute the harmonic function based on a Gaussian random field model. Labeled and unlabeled data are represented as vertices in a weighted graph, with edge weights encoding the similarity between instances. The mean of the field is characterized in terms of harmonic functions.

### Functions:

#### harmonic\_function(G, max\_iter=30, label\_name='label')

#### Parameters:

#### G (NetworkX Graph)

#### max\_iter (int) – maximum number of iterations allowed

#### label\_name (string) – name of target labels to predict

#### Returns:

#### predicted – Array of predicted labels

#### Return type:

#### array, shape = [n\_samples]

#### Raises:

#### NetworkXError – If no nodes on G has label\_name.

## Local and Global Consistency

### Overview:

##### Proposed by Zhu et al. , local and global consistency performed well in classification under the assumption of **consistency**, which means: (1) nearby points are likely to have the same label; and (2) points on the same structure are likely to have the same label.

### Functions:

#### local\_and\_global\_consistency(G, alpha=0.99, max\_iter=30, label\_name='label')

#### Parameters:

#### G (NetworkX Graph)

#### alpha (float) – Clamping factor

#### max\_iter (int) – Maximum number of iterations allowed

#### label\_name (string) – Name of target labels to predict

#### Returns:

#### predicted – Array of predicted labels

#### Return type:

#### array, shape = [n\_samples]

#### Raises:

#### NetworkXError – If no nodes on G has label\_name.

1. https://pypi.org/project/networkx/ [↑](#footnote-ref-0)
2. https://www.sci.unich.it/~francesc/teaching/network/eigenvector.html [↑](#footnote-ref-1)
3. https://aksakalli.github.io/2017/07/17/network-centrality-measures-and-their-visualization.html#hits-hubs-and-authorities [↑](#footnote-ref-2)
4. https://programminghistorian.org/en/lessons/common-similarity-measures [↑](#footnote-ref-3)
5. http://www.scholarpedia.org/article/Similarity\_measures [↑](#footnote-ref-4)
6. https://www.cs.cmu.edu/~jingx/docs/DBreport.pdf [↑](#footnote-ref-5)
7. http://www.cs.uoi.gr/~pvassil/downloads/GraphDistance/LauraZager.pdf [↑](#footnote-ref-6)