Graph&AI

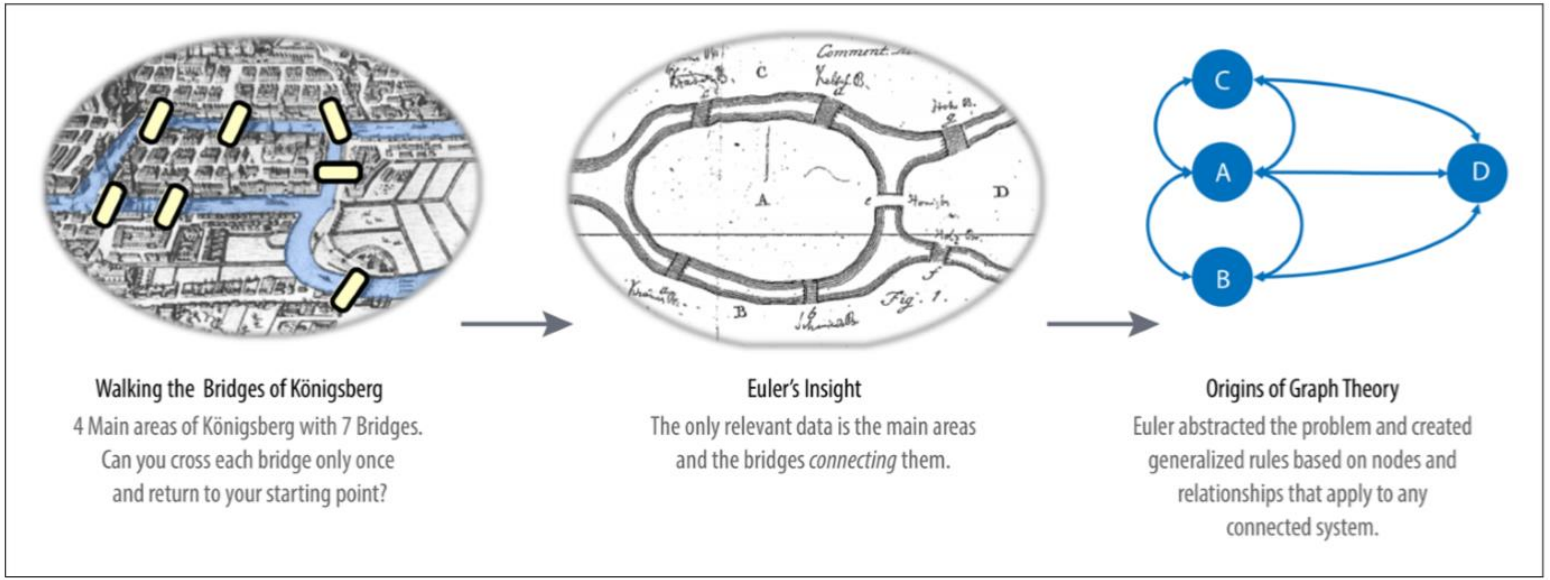
Network Graph Algorithms 2023-24

Path Finding – Community – Centrality

terri.hoare@dbs.ie

Origins of Mathematics of Graph Theory

"Solutio problematis ad geometriam situs pertinentis" – Leonard Euler 1736



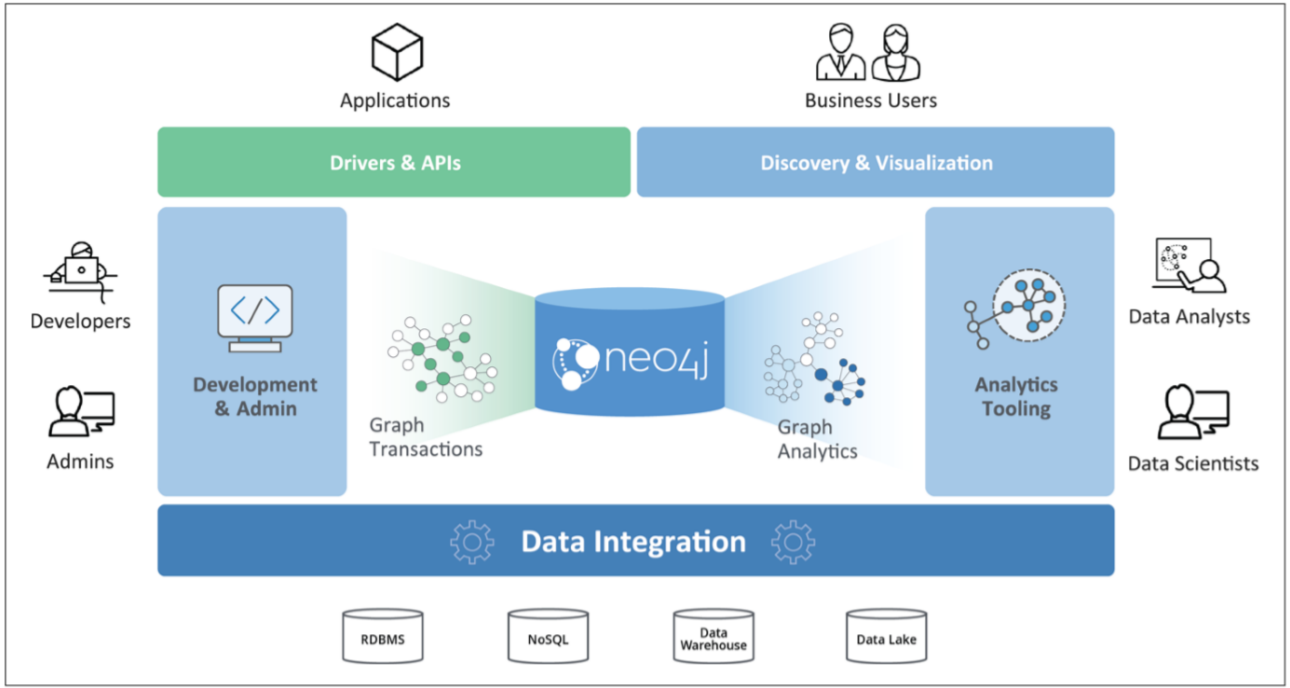
Gartner Top Tech Trends in Data and Analytics 2021

Trend 8: Graph relates everything

Graph forms the foundation of modern data and analytics with capabilities to enhance and improve user collaboration, machine learning models and explainable AI. Although graph technologies are not new to data and analytics, there has been a shift in the thinking around them as organizations identify an increasing number of use cases. In fact, as many as 50% of Gartner client inquiries around the topic of AI involve a discussion around the use of graph technology.

https://arxiv.org/pdf/1806.01261.pdf

Neo4j Graph Database Platform Announced Graph Connect September 2017



DeepMind; Google Brain; MIT; Edinburgh Univ.

Oct. 2018

“Artificial intelligence (AI) has undergone a renaissance recently, making major progress in key domains such as vision, language, control, and decision-making. This has been due, in part, to cheap data and cheap compute resources, which have fit the natural strengths of deep learning.

However, many defining characteristics of human intelligence, which developed under much different pressures, remain out of reach for current approaches. In particular, generalizing beyond one’s experiences—a hallmark of human intelligence from infancy—remains a formidable challenge for modern AI…

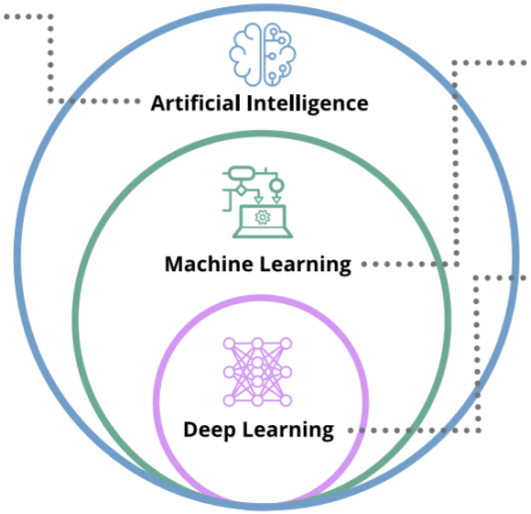
We discuss how graph networks can support relational reasoning and combinatorial generalization, laying the foundation for more sophisticated, interpretable, and flexible patterns of reasoning.”

https://arxiv.org/pdf/1806.01261.pdf

Neo4j and AI

Artificial Intelligence (AI)
A computer process that has learned to solve tasks in a way that mimics human decisions

AI solutions today are mostly used for very specific tasks, versus general applications

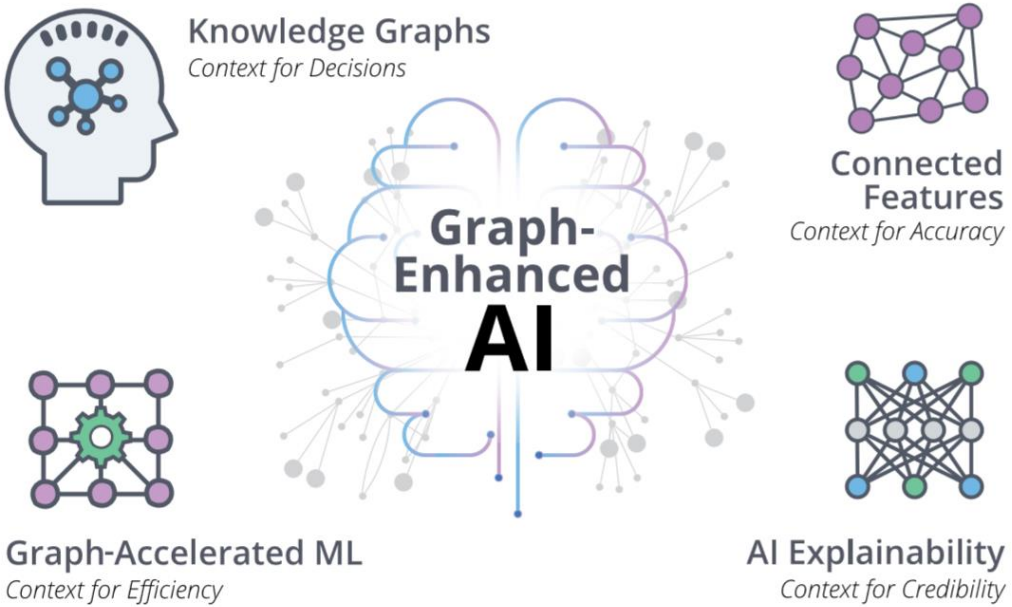


Machine Learning (ML)
Uses algorithms to help computers learn by task-specific examples and progressive improvements, without explicit programming

Deep Learning (DL)
Uses a cascade of processing layers modeled on neural networks to learn data representations such as features or classifications

Artificial intelligence consists of several subsets of technologies that each solve problems in different ways.

Neo4j and AI


Knowledge Graphs
Context for Decisions
Connected Features
Context for Accuracy
Graph-Enhanced AI
Graph-Accelerated ML
Context for Efficiency
AI Explainability
Context for Credibility

Network Science and Graph Analytics

Types of questions Graph Analytics answer

- Investigate the route of a disease or a cascading transport failure.
- Uncover the most vulnerable, or damaging, components in a network attack.
- Identify the least costly or fastest way to route information or resources.
- Predict missing links in your data.
- Locate direct and indirect influence in a complex system.
- Discover unseen hierarchies and dependencies.
- Forecast whether groups will merge or break apart.
- Find bottlenecks or who has the power to deny/provide more resources.
- Reveal communities based on behaviour for personalized recommendations.
- Reduce false positives in fraud and anomaly detection.
- Extract more predictive features for machine learning.

Network Science and Graph Algorithms

Most graph queries consider specific parts of the graph, and the work is usually focused in the surrounding subgraph. We term this type of work graph local. This type of graph-local processing is often utilized for real-time transactions and pattern-based queries.

When speaking about graph algorithms, we are typically looking for global patterns and structures. The input to the algorithm is usually the whole graph, and the output can be an enriched graph or some aggregate value such as a score. This approach sheds light on the overall nature of a network through its connections. Organizations tend to use graph algorithms to model systems and predict behaviour based on how things disseminate, important components, group identification, and the overall robustness of the system.

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Graph Algorithm Categories

Community Detection

Detects group clustering or partition options

Centrality / Importance

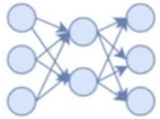
Determines the importance of distinct nodes in the network

Pathfinding & Search

Finds optimal paths or evaluates route availability and quality

Heuristic Link Prediction

Estimates the likelihood of nodes forming a relationship



Embeddings

Vectors that capture connectivity or topology

Similarity

Evaluates how alike nodes are

Pathfinding and Search Algorithms

Graph search algorithms explore a graph either for general discovery or explicit search. These algorithms carve paths through the graph, but there is no expectation that those paths are computationally optimal. We will cover Breadth First Search and Depth First Search because they are fundamental for traversing a graph and are often a required first step for many other types of analysis.

Pathfinding algorithms build on top of graph search algorithms and explore routes between nodes, starting at one node and traversing through relationships until the
destination has been reached. These algorithms are used to identify optimal routes through a graph for uses such as logistics planning, least cost call or IP routing, and gaming simulation.

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Pathfinding and Search Algorithms

Pathfinding algorithms include:

All Pairs Shortest Path and Single Source Shortest Path

For finding the shortest paths between all pairs or from a chosen node to all others

Minimum Spanning Tree

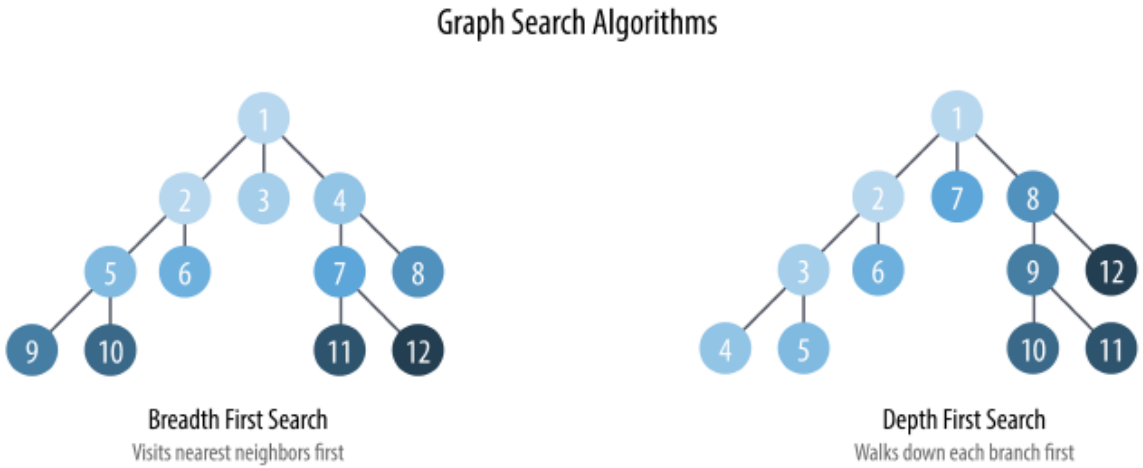
For finding a connected tree structure with the smallest cost for visiting all nodes
from a chosen node

Random Walk

A useful pre-processing/sampling step for machine learning workflows and other graph algorithms

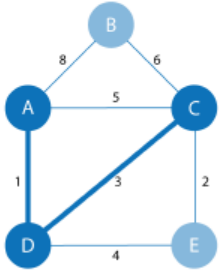
Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Pathfinding and Search Algorithms



Pathfinding and Search Algorithms

Pathfinding Algorithms



Calculated by relationship weights

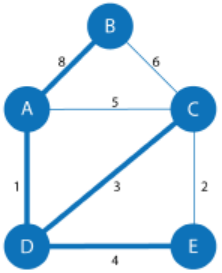
Shortest Path

Shortest path between 2 nodes (A to C shown)

(A, B) = 8
(A, C) = 4 via D
(A, D) = 1
(A, E) = 5 via D
(B, C) = 6
(B, D) = 9 via A or C
And so on...

All-Pairs Shortest Paths

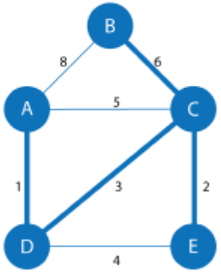
Optimized calculations for shortest paths from **all** nodes to **all** other nodes



Single Source Shortest Path

Shortest path from **a root** node (A shown) to **all** other nodes

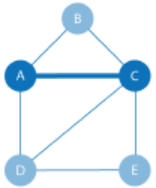
Traverses to the next unvisited node via the lowest cumulative weight **from the root**



Minimum Spanning Tree

Shortest path **connecting all nodes** (A start shown)

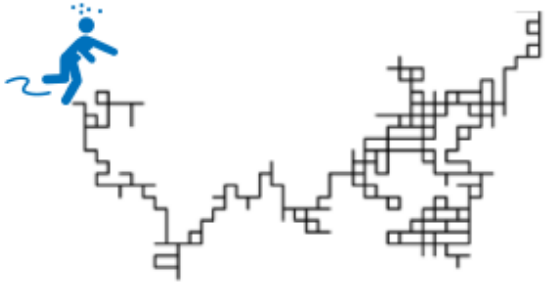
Traverses to the next unvisited node via the lowest weight **from any visited node**



Calculated by number of hops

Pathfinding and Search Algorithms

Random Pathfinding Algorithm



Random Walk

Provides a set of random, connected nodes by following any relationship, selected somewhat randomly

Also called the drunkard's walk

Pathfinding and Search Algorithms

Algorithm type	What it does	Example use	Spark example	Neo4j example
Breadth First Search	Traverses a tree structure by fanning out to explore the nearest neighbors and then their sublevel neighbors	Locating neighbor nodes in GPS systems to identify nearby places of interest	Yes	No
Depth First Search	Traverses a tree structure by exploring as far as possible down each branch before backtracking	Discovering an optimal solution path in gaming simulations with hierarchical choices	No	No
Shortest Path Variations: A*, Yen's	Calculates the shortest path between a pair of nodes	Finding driving directions between two locations	Yes	Yes
All Pairs Shortest Path	Calculates the shortest path between *all pairs of nodes* in the graph	Evaluating alternate routes around a traffic jam	Yes	Yes
Single Source Shortest Path	Calculates the shorest path between a *single root* node and *all* other nodes	Least cost routing of phone calls	Yes	Yes
Minimum Spanning Tree	Calculates the path in a connected tree structure with the smallest cost for visiting all nodes	Optimizing connected routing, such as laying cable or garbage collection	No	Yes
Random Walk	Returns a list of nodes along a path of specified size by randomly choosing relationships to traverse.	Augmenting training for machine learning or data for graph algorithms.	No	Yes

Pathfinding and Search Algorithms

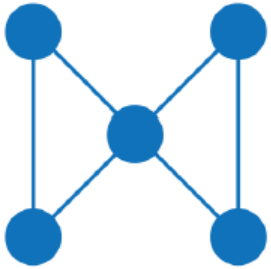
There are two special paths in graph analysis that are worth noting. First, a ***Eulerian path*** is one where every relationship is visited exactly once. Second, a ***Hamiltonian path*** is one where every node is visited exactly once. A path can be both Eulerian and Hamiltonian, and if you start and finish at the same node it's considered a *cycle* or *tour*.

The Konigsberg bridges problem was searching for a Eulerian cycle. This applies to routing scenarios such as mail delivery. The Hamiltonian cycle is best known from its relation to the *Traveling Salesman Problem* (TSP), which asks, "What's the shortest possible route for a salesperson to visit each of their assigned cities and return to the origin city?" While similar to the Eulerian tour, the TSP is computationally more intensive with approximation alternatives. It is an NP-Hard problem in combinatorial optimisation important in theoretical computer science and operations research. It's used in a wide variety of planning, logistics, and optimization problems.

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

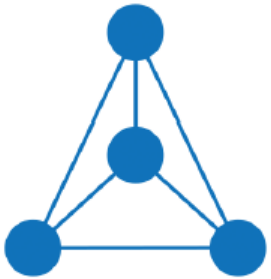
Pathfinding and Search Algorithms

Eulerian Cycle
not Hamiltonian



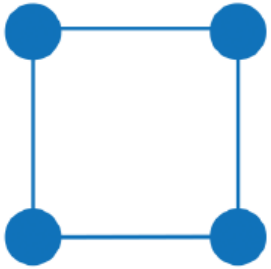
Visit every relationship once
(allowed to repeat nodes)

Hamiltonian Cycle
not Eulerian



Visit every node once
(allowed to repeat relationships)

Eulerian and Hamiltonian



Shortest Path Algorithm

The Shortest Path algorithm calculates the shortest (weighted) path between a pair of nodes. It's useful for user interactions and dynamic workflows because it works in real time. Pathfinding has a history dating back to the 19th century and is considered to be a classic graph problem. It gained prominence in the early 1950s in the context of alternate routing; that is, finding the second-shortest route if the shortest route is blocked. In 1956, Edsger Dijkstra created the best-known of these algorithms. Dijkstra's Shortest Path algorithm operates by first finding the lowest-weight relationship from the start node to directly connected nodes. It keeps track of those weights and moves to the "closest" node. It then performs the same calculation, but now as a cumulative total from the start node. The algorithm continues to do this, evaluating a "wave" of cumulative weights and always choosing the lowest weighted cumulative path to advance along, until it reaches the destination node.

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Dijkstra's Shortest Path Algorithm in Concept

Let distance of start vertex from start vertex = 0
Let distance of all other vertices from start = ∞
Repeat
  Visit the unvisited vertex with the smallest known distance from the start vertex
  For the current vertex, examine its unvisited neighbours
  For the current vertex, calculate distance of each neighbour from start vertex
  If the calculated distance of the vertex is less than the known distance, update the shortest distance
  Update the previous vertex for each of the updated distances
  Add the current vertex to the list of updated vertices
Until all vertices finished

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Dijkstra's Shortest Path Algorithm Formalism

Let G be a directed network with vertices V = {1, . . . ,n} such that, for each arc ij, its length $d_{ij} > 0$.

The essence of Dijkstra's algorithm is a labelling procedure.

At a typical stage of the algorithm we have V = P ∪ S, where

- P is the set of permanently labelled vertices, and
- S is the set of temporarily (or tentatively) labelled vertices.

To begin with, S = V , then as the algorithm proceeds, S gets smaller, while P gets bigger until eventually

all vertices have received permanent labels and the algorithm terminates.

In fact, each vertex u will have two labels:

- D_u, our current guess at the distance from 1 to u;
- p_u, our current guess at the parent of u

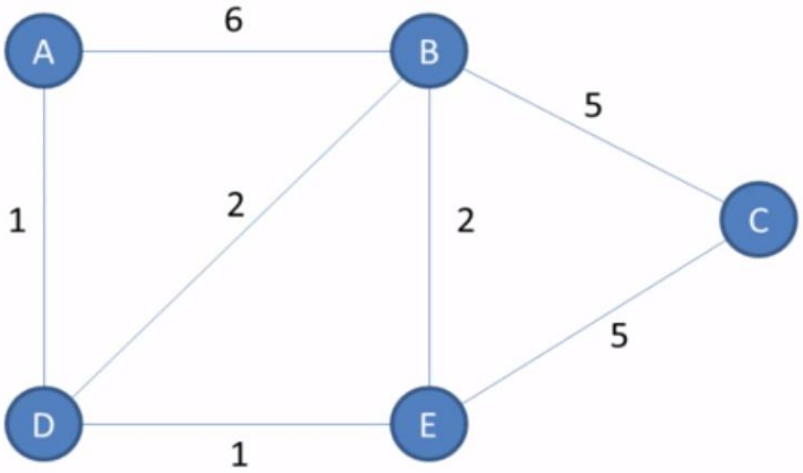
Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Dijkstra's Shortest Path Algorithm Pseudo Code

Algorithm 5.2: *Dijkstra*(r, G, T)

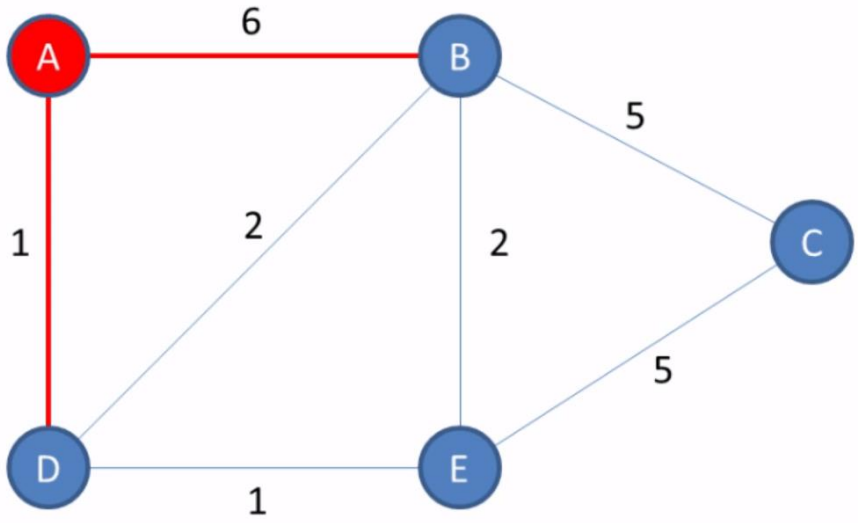
$P := \{1\}$; $S := V \setminus P = \{2, \ldots, n\}$; $A := \varnothing$; // *initialise* P, S *and* A
$D_1 := 0$;
for *each vertex* $u \in S$ **do**
  $D_u := d_{1u}$; $p_u := 1$; // *initialise* D *and* p *labels*
end for
while $S \neq \varnothing$ **do**
  Select $u \in S$ with $D_u = \min_{v \in S} D_v$;
  $P := P \cup \{u\}$; $S := S \setminus \{u\}$; // *move* u *from* S *to* P
  $A := A \cup \{(p_u, u)\}$;
  for *all* $v \in S$ **do** // *note* u *is no longer in* S
    if $D_v > D_u + d_{uv}$ **then**
      $D_v := D_u + d_{uv}$; // *relax inequality*
      $p_v := u$; // *set the parent of* v *to be* u
    end if
  end for
end while

Dijkstra's Shortest Path Algorithm Solution



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Dijkstra's Shortest Path Algorithm Worked Example

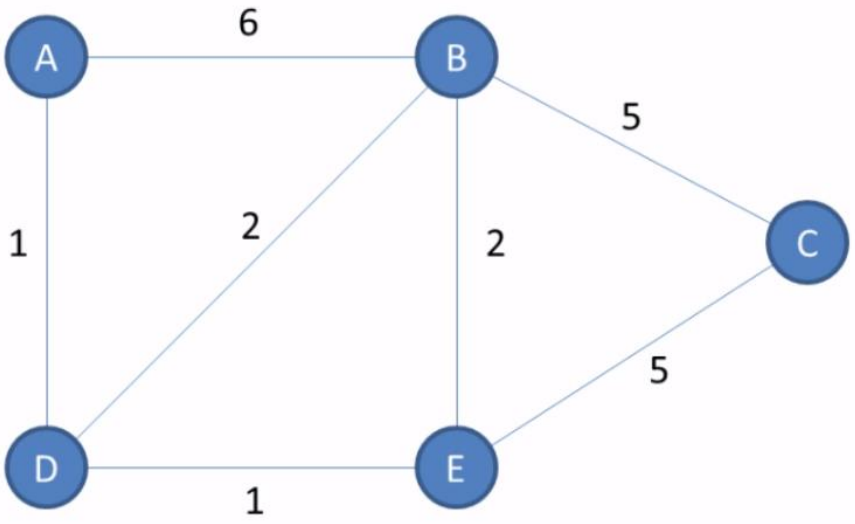


Vertex	Shortest distance from A	Previous vertex
A	0	
B	∞	
C	∞	
D	∞	
E	∞	

Visited = [ ]

Unvisited = [A, B, C, D, E]

Dijkstra's Shortest Path Algorithm Worked Example

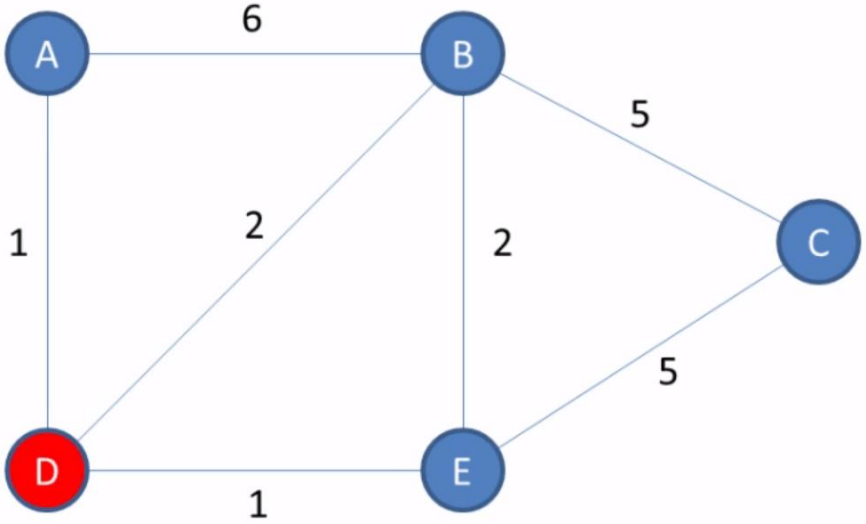


Vertex	Shortest distance from A	Previous vertex
A	0	
B	6	A
C	∞	
D	1	A
E	∞	

Visited = [A]

Unvisited = [B, C, D, E]

Dijkstra's Shortest Path Algorithm Worked Example

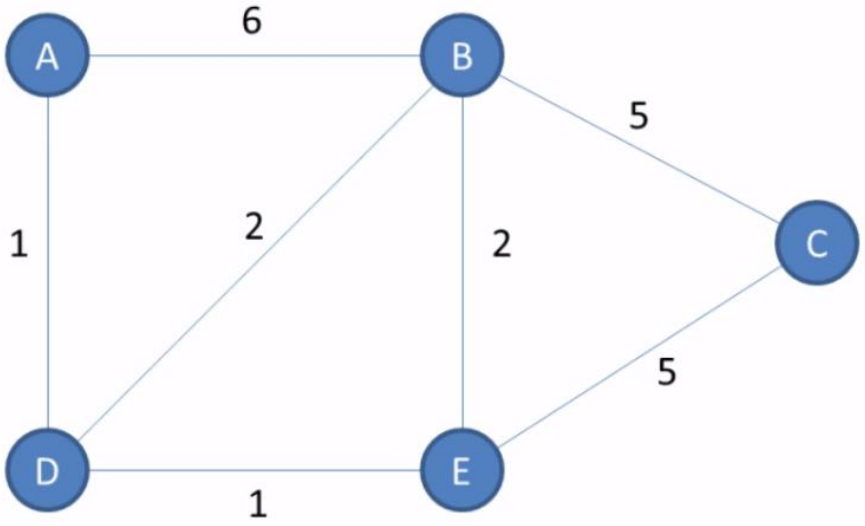


Vertex	Shortest distance from A	Previous vertex
A	0	
B	6	A
C	∞	
D	1	A
E	∞	

Visited = [A]

Unvisited = [B, C, D, E]

Dijkstra's Shortest Path Algorithm Worked Example

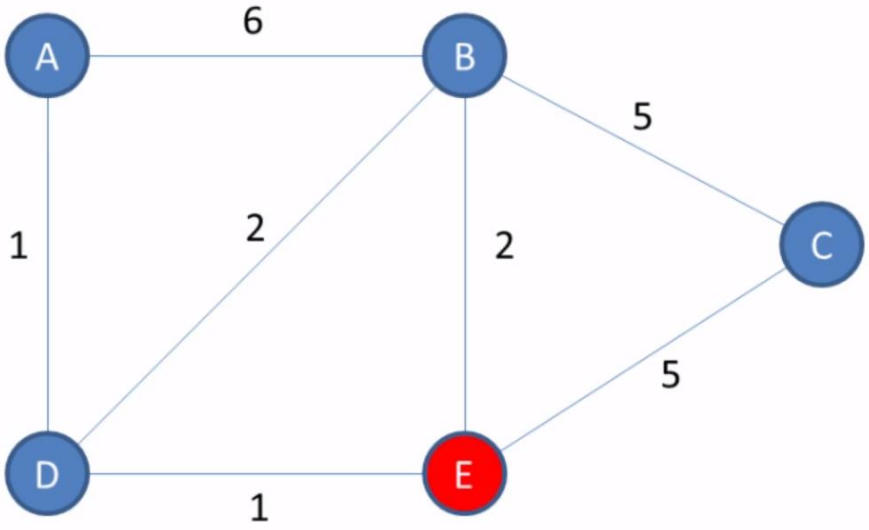


Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	∞	
D	1	A
E	2	D

Visited = [A, D]

Unvisited = [B, C, E]

Dijkstra's Shortest Path Algorithm Worked Example

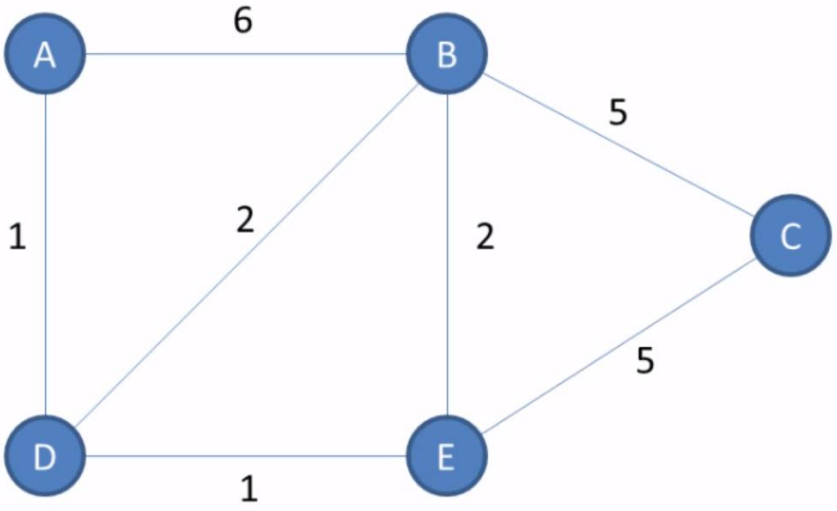


Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	∞	
D	1	A
E	2	D

Visited = [A, D]

Unvisited = [B, C, E]

Dijkstra's Shortest Path Algorithm Worked Example

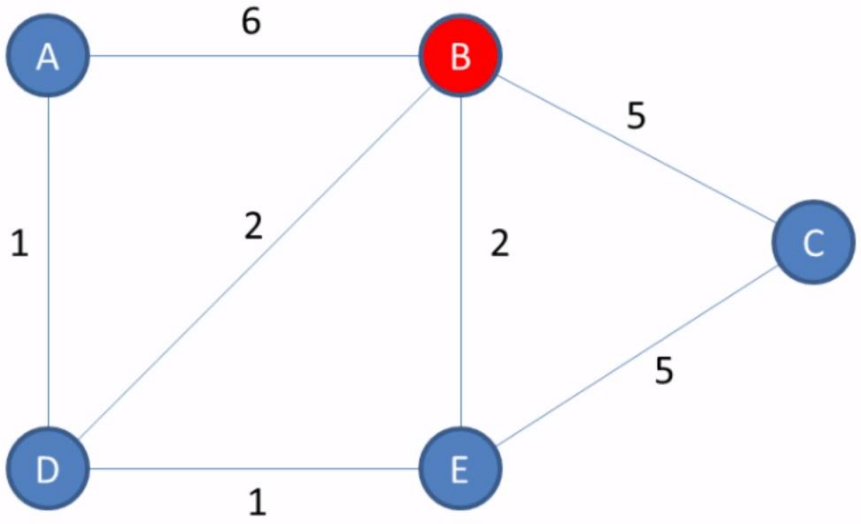


Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Visited = [A, D, E]

Unvisited = [B, C]

Dijkstra's Shortest Path Algorithm Worked Example

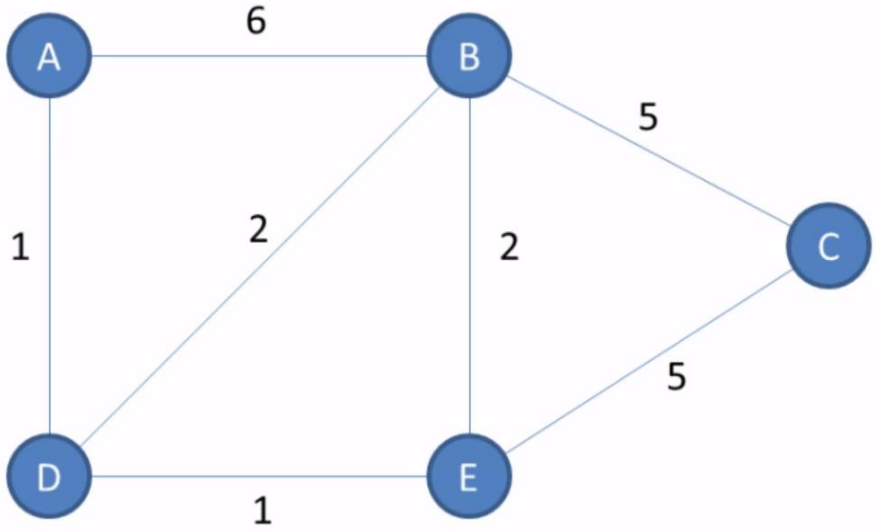


Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Visited = [A, D, E]

Unvisited = [B, C]

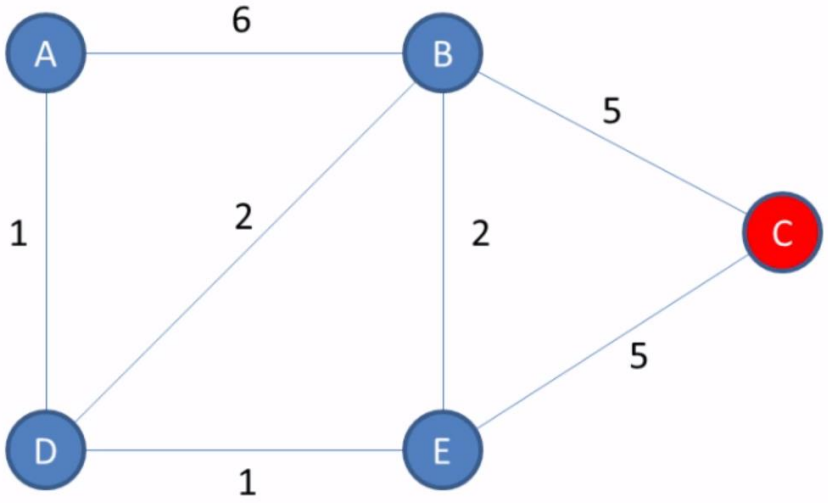
Dijkstra's Shortest Path Algorithm Worked Example



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Visited = [A, D, E, B] Unvisited = [C]

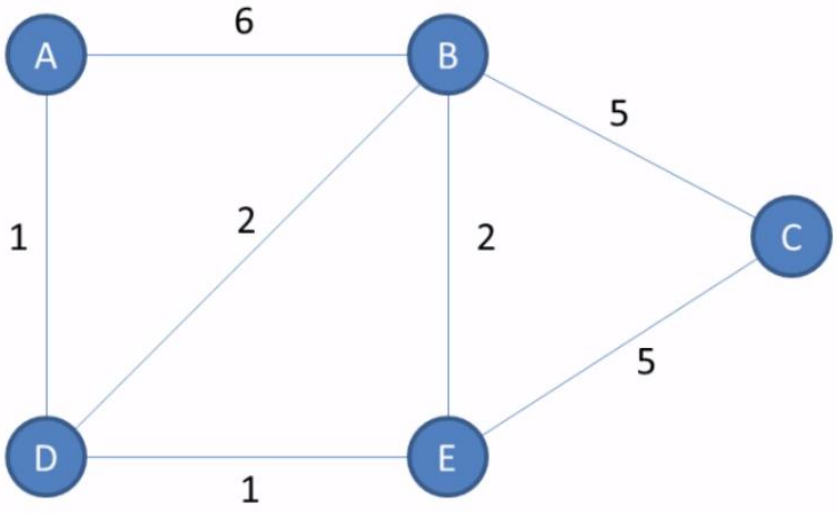
Dijkstra's Shortest Path Algorithm Worked Example



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Visited = [A, D, E, B] Unvisited = [C]

Dijkstra's Shortest Path Algorithm Worked Example



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Visited = [A, D, E, B, C] Unvisited = []

Shortest Path Algorithm Use Cases

Use Shortest Path to find optimal routes between a pair of nodes, based on either the number of hops or any weighted relationship value. For example, it can provide real-time answers about degrees of separation, the shortest distance between points, or the least expensive route. You can also use this algorithm to simply explore the connections between particular nodes.

- Finding directions between locations. Web-mapping tools such as Google Maps use the Shortest Path algorithm, or a close variant, to provide driving directions.
- Finding the degrees of separation between people in social networks. For example, when you view someone's profile on LinkedIn, it will indicate how many people separate you in the graph, as well as listing your mutual connections.

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Shortest Path Algorithm Use Cases

Use Shortest Path to find optimal routes between a pair of nodes, based on either the number of hops or any weighted relationship value. For example, it can provide real-time answers about degrees of separation, the shortest distance between points, or the least expensive route. You can also use this algorithm to simply explore the connections between particular nodes.

- Finding directions between locations. Web-mapping tools such as Google Maps use the Shortest Path algorithm, or a close variant, to provide driving directions.
- Finding the degrees of separation between people in social networks. For example, when you view someone's profile on LinkedIn, it will indicate how many people separate you in the graph, as well as listing your mutual connections.

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Shortest Path Algorithm Use Cases

- Finding the number of degrees of separation between an actor and Kevin Bacon based on the movies they've appeared in (the *Bacon Number*). An example of this can be seen on the Oracle of Bacon website. The Erdos Number Project provides a similar graph analysis based on collaboration with Paul Erdos, one of the most prolific mathematicians of the twentieth century.

Shortest Path Variation A*

The A* Shortest Path algorithm improves on Dijkstra's by finding shortest paths more quickly. It does this by allowing the inclusion of extra information that the algorithm can use, as part of a heuristic function, when determining which paths to explore next. The algorithm was invented by Peter Hart, Nils Nilsson, and Bertram Raphael and described in their 1968 paper "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". The A* algorithm operates by determining which of its partial paths to expand at each iteration of its main loop. It does so based on an estimate of the cost (heuristic) still left to reach the goal node.

A* selects the path that minimizes the function, $f(n) = g(n) + h(n)$ where $g(n)$ is the cost of the path from the starting point to node n and $h(n)$ is the estimated cost of the path from node n to the destination node, as computed by a heuristic.

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Shortest Path Variation Yen's k-Shortest Paths

Yen's *k*-Shortest Paths algorithm is similar to the Shortest Path algorithm, but rather than finding just the shortest path between two pairs of nodes, it also calculates the second shortest path, third shortest path, and so on up to *k*-1 deviations of shortest paths.

Jin Y. Yen invented the algorithm in 1971 and described it in "Finding the *K* Shortest Loopless Paths in a Network". This algorithm is useful for getting alternative paths when finding the absolute shortest path isn't our only goal. It can be particularly helpful when we need more than one backup plan!

Community Detection Algorithms

Introduction

Community formation is common in all types of networks and identifying them is essential for evaluating group behaviour and emergent phenomena. The general principle in finding communities is that its members will have more relationships within the group than with nodes outside their group. Identifying these related sets reveals clusters of nodes, isolated groups, and network structure. This information helps infer similar behaviour or preferences of peer groups, estimate resiliency, find nested relationships, and prepare data for other analyses. Community detection algorithms are also commonly used to produce network visualization for general inspection.

We use the terms *set*, *partition*, *cluster*, *group*, and *community* interchangeably. These terms are different ways to indicate that similar nodes can be grouped. Community detection algorithms are also called clustering and partitioning algorithms.

Community Detection Algorithms

Introduction

We'll provide details on the most representative community detection algorithms:

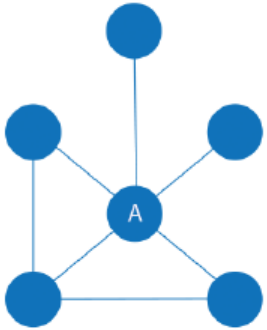
- **Triangle Count** and **Clustering Coefficient** for overall relationship density
- **Strongly Connected Components** and **Weakly Connected Components** for finding connected clusters
- **Label Propagation** for quickly inferring groups based on node labels
- **Louvain Modularity** for looking at grouping quality and hierarchies

*** We use weighted relationships for these algorithms because they're typically used to capture the significance of different relationships.

Community Detection Algorithms

Types of Algorithms

Measuring Algorithms



Triangle Count

The number of triangles that pass through a node. A has two triangles.

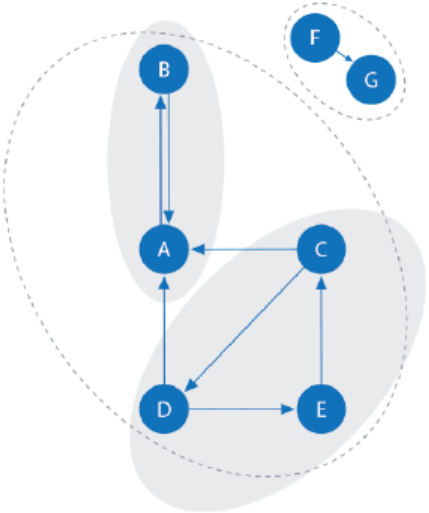
Clustering Coefficient

The probability that the neighbors of a node are connected to each other.

A has a 0.2 CC. Any 2 nodes connected to A have a 20% chance of being connected to each other.

These measures can be counted/normalized globally.

Components Algorithms



Connected Components

Sets where all nodes can reach all other nodes, regardless of direction.

2 sets shown with dashed outlines: {A,B,C,D,E} and {F,G}.

Strongly Connected Components

Sets where all nodes can reach all other nodes in both directions, but not necessarily directly.

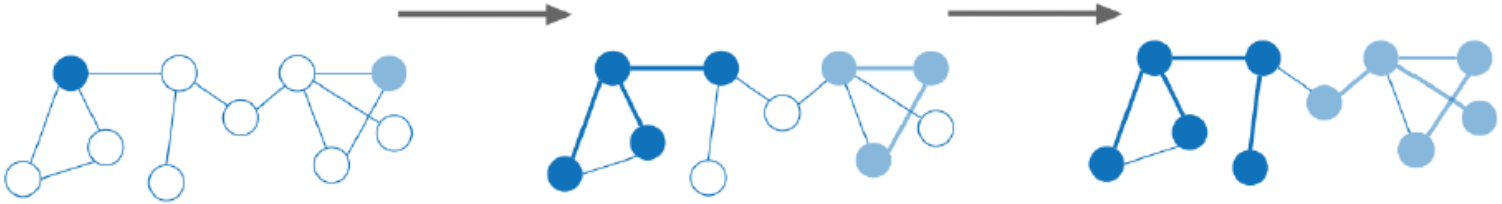
2 sets shown shaded:

Community Detection Algorithms

Types of Algorithms

Label Propagation Algorithm

Spread labels to or from neighbors to find clusters.



Run over multiple iterations.

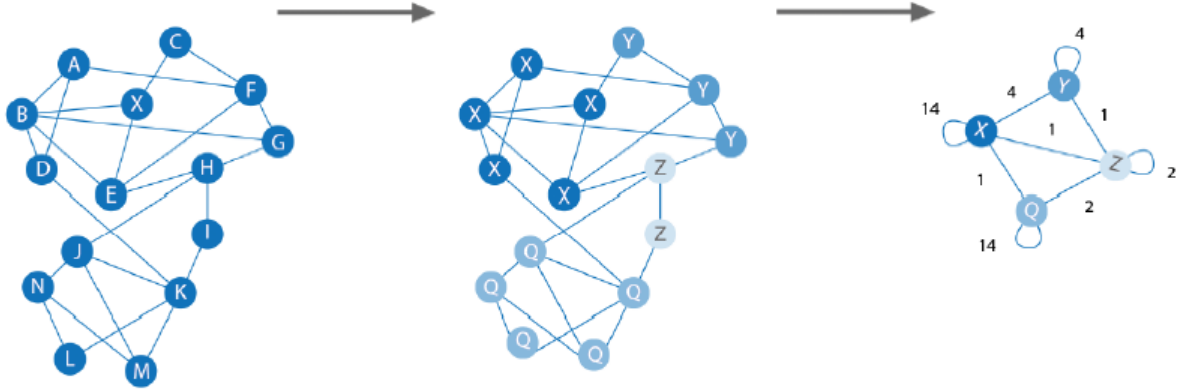
Weights of relationships and/or nodes are often used to determine label "popularity" in a group.

Community Detection Algorithms

Types of Algorithms

Louvain Modularity Algorithm

Find clusters by moving nodes into higher relationship density groups and aggregating into supercommunities.



Run over multiple iterations.

Relationship weights and totals are used to determine grouping.

Community Detection Algorithms Overview

Algorithm type	What it does	Example use	Spark example	Neo4j example
Triangle Count and Clustering Coefficient	Measures how many nodes form triangles and the degree to which nodes tend to cluster together	Estimating group stability and whether the network might exhibit "small-world" behaviors seen in graphs with tightly knit clusters	Yes	Yes
Strongly Connected Components	Finds groups where each node is reachable from every other node in that same group *following the direction* of relationships	Making product recommendations based on group affiliation or similar items	Yes	Yes
Connected Components	Finds groups where each node is reachable from every other node in that same group, *regardless of the direction* of relationships	Performing fast grouping for other algorithms and identify islands	Yes	Yes
Label Propagation	Infers clusters by spreading labels based on neighborhood majorities	Understanding consensus in social communities or finding dangerous combinations of possible co-prescribed drugs	Yes	Yes
Louvain Modularity	Maximizes the presumed accuracy of groupings by comparing relationship weights and densities to a defined estimate or average	In fraud analysis, evaluating whether a group has just a few discrete bad behaviors or is acting as a fraud ring	No	Yes

Community Detection Algorithms

Application Hint

When using community detection algorithms, be conscious of the density of the relationships. If the graph is very dense, you may end up with all nodes congregating
in one or just a few clusters. You can counteract this by filtering by degree, relationship weights, or similarity metrics. On the other hand, if the graph is too sparse with few connected nodes, you may end up with each node in its own cluster. In this case, try to incorporate additional relationship types that carry more relevant information.

Community Detection Algorithms

Example Graph Data – Software Dependency Graph

Dependency graphs are particularly well suited for demonstrating the sometimes subtle differences between community detection algorithms because they tend to be more connected and hierarchical. The examples in this chapter are run against a graph containing dependencies between Python libraries, although dependency graphs are used in various fields, from software to energy grids. This kind of software dependency graph is used by developers to keep track of transitive interdependencies and conflicts in software projects.

Community Detection Algorithms

Example Graph Data – Software Dependency Graph

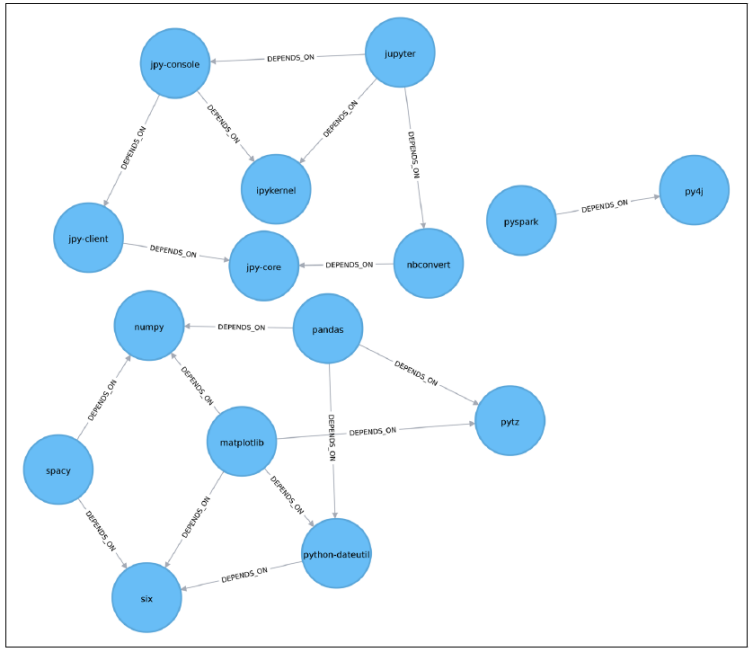
id
six
pandas
numpy
python-dateutil
pytz
pyspark
matplotlib
spacy
py4j
jupyter
jpy-console
nbconvert
ipykernel
jpy-client
jpy-core

src	dst	relationship
pandas	numpy	DEPENDS_ON
pandas	pytz	DEPENDS_ON
pandas	python-dateutil	DEPENDS_ON
python-dateutil	six	DEPENDS_ON
pyspark	py4j	DEPENDS_ON
matplotlib	numpy	DEPENDS_ON
matplotlib	python-dateutil	DEPENDS_ON
matplotlib	six	DEPENDS_ON

src	dst	relationship
matplotlib	pytz	DEPENDS_ON
spacy	six	DEPENDS_ON
spacy	numpy	DEPENDS_ON
jupyter	nbconvert	DEPENDS_ON
jupyter	ipykernel	DEPENDS_ON
jupyter	jpy-console	DEPENDS_ON
jpy-console	jpy-client	DEPENDS_ON
jpy-console	ipykernel	DEPENDS_ON
jpy-client	jpy-core	DEPENDS_ON
nbconvert	jpy-core	DEPENDS_ON

Community Detection Algorithms

Software Dependency Graph Model



There are three clusters of libraries. We can use visualizations on smaller datasets as a tool to help validate the clusters derived by community detection algorithms.

Community Detection Algorithms

Triangle Count and Clustering Coefficient

The Triangle Count and Clustering Coefficient algorithms are presented together because they are so often used together. Triangle Count determines the number of triangles passing through each node in the graph. A triangle is a set of three nodes, where each node has a relationship to all other nodes. Triangle Count can also be run globally for evaluating our overall dataset. Networks with a high number of triangles are more likely to exhibit small-world structures and behaviours.

Community Detection Algorithms

Triangle Count and Clustering Coefficient

The goal of the Clustering Coefficient algorithm is to measure how tightly a group is clustered compared to how tightly it could be clustered. The algorithm uses Triangle Count in its calculations, which provides a ratio of existing triangles to possible relationships. A maximum value of 1 indicates a clique where every node is connected to every other node. There are two types of clustering coefficients: local clustering and global clustering.

Community Detection Algorithms

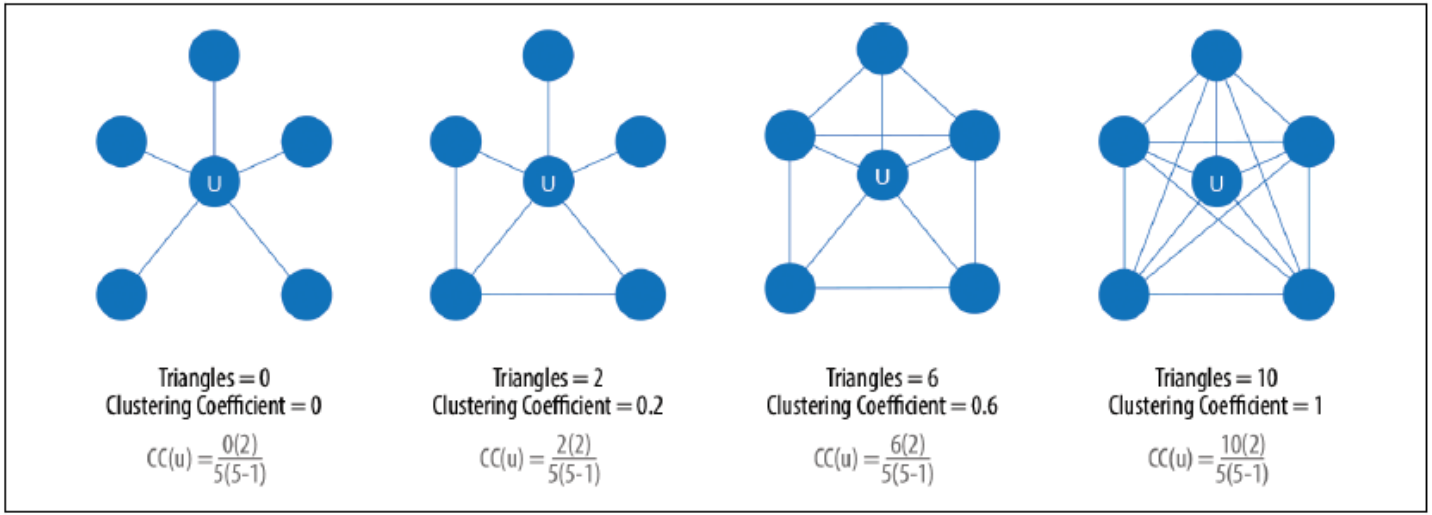
Local Clustering Coefficient

The local clustering coefficient of a node is the likelihood that its neighbours are also connected. The computation of this score involves triangle counting. The clustering coefficient of a node can be found by multiplying the number of triangles passing through the node by two and then diving that by the maximum number of relationships in the group, which is always the degree of that node, minus one.

*** Refer Table next slide, we use a node with five relationships which makes it appear that the clustering coefficient will always equate to 10% of the number of triangles. We can see this is not the case when we alter the number of relationships. If we change the second example to have four relationships (and the same two triangles) then the coefficient is 0.33.

Community Detection Algorithms

Local Clustering Coefficient



Community Detection Algorithms

Local Clustering Coefficient - Formula

$$CC(u) = \frac{2R_u}{k_u(k_u - 1)}$$

where
:
u is a node

$R(u)$ is the number of relationships through the neighbours of u (this can be obtained by using the number of triangles passing through u)

$k(u)$ is the degree of u

Community Detection Algorithms

Global Clustering Coefficient

The global clustering coefficient is the normalized sum of the local clustering coefficients. Clustering coefficients give us an effective means to find obvious groups like cliques, where every node has a relationship with all other nodes, but we can also specify thresholds to set levels (say, where nodes are 40% connected).

Community Detection Algorithms

When to Use Triangle Count and Clustering Coefficients?

Use **Triangle Count** when you need to determine the stability of a group or as part of calculating other network measures such as the clustering coefficient. Triangle counting is popular in social network analysis, where it is used to detect communities.

Clustering Coefficient can provide the probability that randomly chosen nodes will be connected. You can also use it to quickly evaluate the cohesiveness of a specific group or your overall network. Together these algorithms are used to estimate resiliency and look for network structures.

Community Detection Algorithms

Triangle Count and Clustering Coefficients – Use Cases

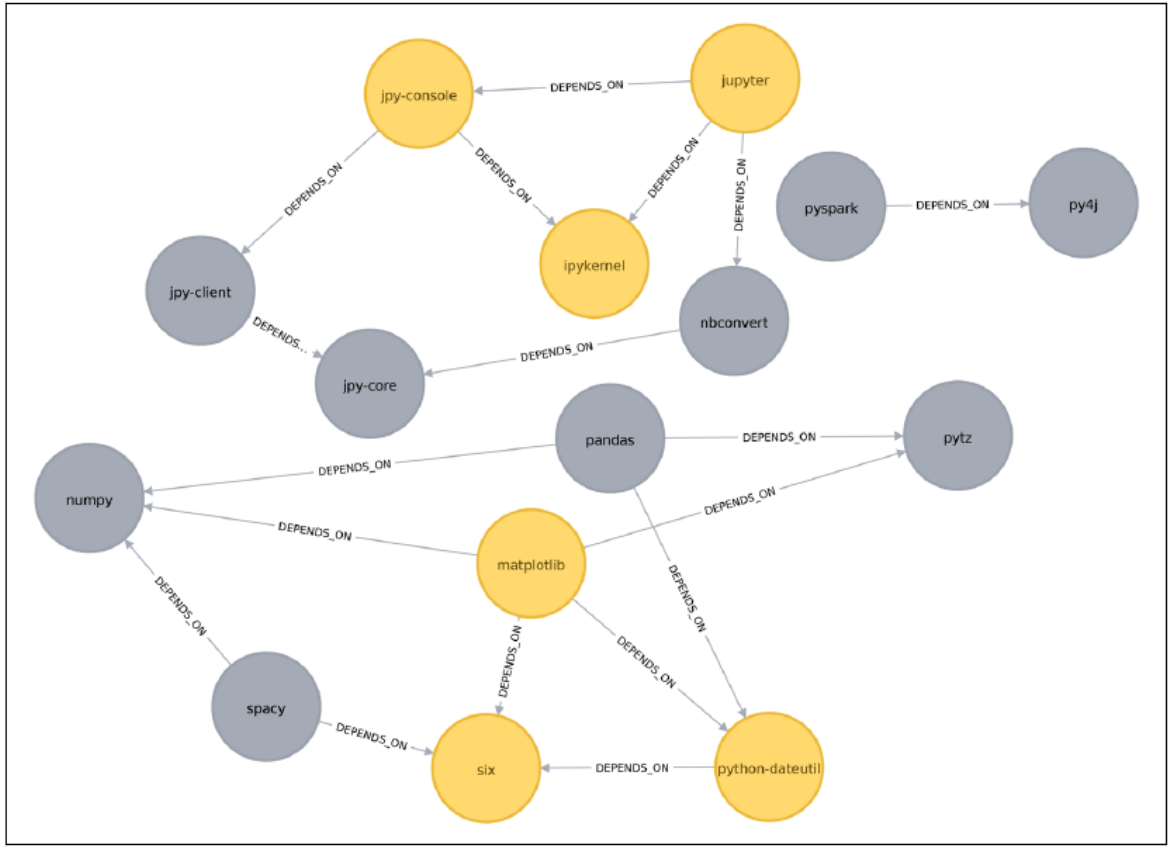
- Identifying features for classifying a given website as spam content. This is described in "Efficient Semi-Streaming Algorithms for Local Triangle Counting in Massive Graphs", a paper by L. Becchetti et al.

- Investigating the community structure of Facebook's social graph, where researchers found dense neighbourhoods of users in an otherwise sparse global graph. Find this study in the paper "The Anatomy of the Facebook Social Graph", by J. Ugander et al.

Community Detection Algorithms

Triangle Count and Clustering Coefficients – Use Cases

- Exploring the thematic structure of the web and detecting communities of pages with common topics based on the reciprocal links between them. For more information, see "Curvature of Co-Links Uncovers Hidden Thematic Layers in the World Wide Web", by J.-P. Eckmann and E. Moses.



Community Detection Algorithms Triangles in the Software Dependency Graph

Community Detection Algorithms

Strongly Connected Components

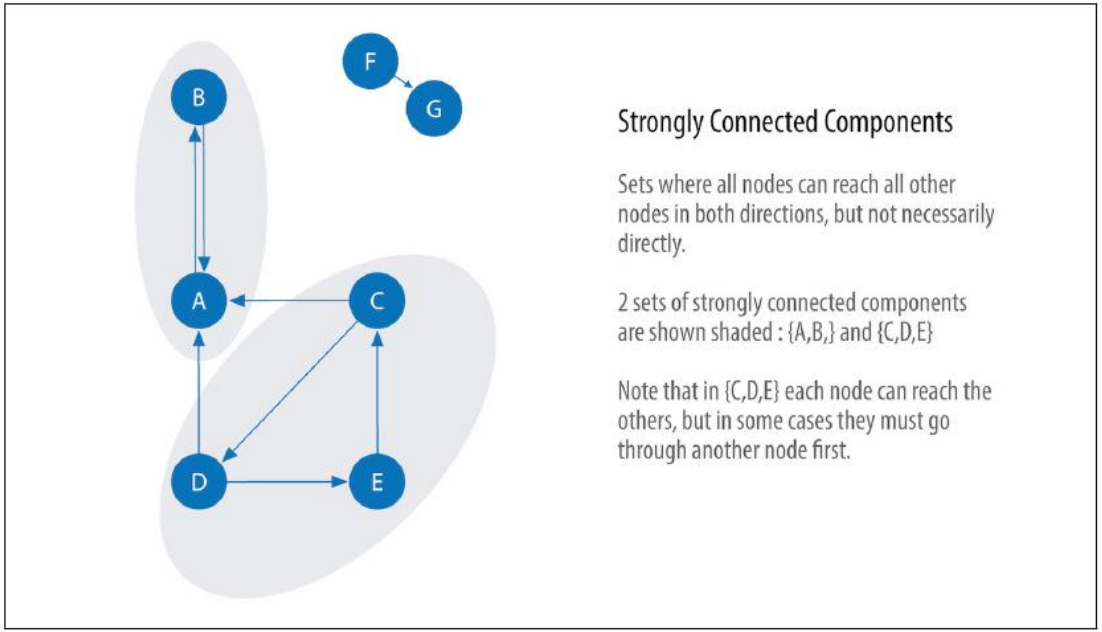
The **Strongly Connected Components (SCC)** algorithm is one of the earliest graph algorithms. SCC finds sets of connected nodes in a directed graph where each node is reachable in both directions from any other node in the same set. Its runtime operations scale well, proportional to the number of nodes. In Figure you can see that the nodes in an SCC group don't need to be immediate neighbours, but there must be directional paths between all nodes in the set.

** Decomposing a directed graph into its strongly connected components is a classic application of the Depth First Search algorithm. Neo4j uses DFS under the hood as part of its implementation of the SCC algorithm.

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Community Detection Algorithms

Strongly Connected Components



Community Detection Algorithms

When to Use Strongly Connected Components

Use **Strongly Connected Components** as an early step in graph analysis to see how a graph is structured or to identify tight clusters that may warrant independent investigation. A component that is strongly connected can be used to profile similar behaviour or inclinations in a group for applications such as recommendation engines.

Many community detection algorithms like SCC are used to find and collapse clusters into single nodes for further inter-cluster analysis. You can also use SCC to visualise cycles for analyses like finding processes that might deadlock because each subprocess is waiting for another member to take an action.

Community Detection Algorithms

Strongly Connected Components – Use Cases

- Finding the set of firms in which every member directly and/or indirectly owns shares in every other member, as in "The Network of Global Corporate Control", an analysis of powerful transnational corporations by S. Vitali, J. B. Glattfelder, and S. Battiston.

- Computing the connectivity of different network configurations when measuring routing performance in multi-hop wireless networks. Read more in "Routing Performance in the Presence of Unidirectional Links in Multihop Wireless Networks", by M. K. Marina and S. R. Das.

Community Detection Algorithms

Strongly Connected Components – Use Cases

- Acting as the first step in many graph algorithms that work only on strongly connected graphs. In social networks we find many strongly connected groups. In these sets people often have similar preferences, and the SCC algorithm is used to find such groups and suggest pages to like or products to purchase to the people in the group who have not yet done so.

Community Detection Algorithms

SCC Application – Software Dependency Graph

partition	libraries
8	[ipykernel]
11	[six]
2	[matplotlib]
5	[jupyter]
14	[python-dateutil]
13	[numpy]
4	[py4j]
7	[nbconvert]
1	[pyspark]
10	[jpy-core]
9	[jpy-client]
3	[spacy]
12	[pandas]
6	[jpy-console]
0	[pytz]

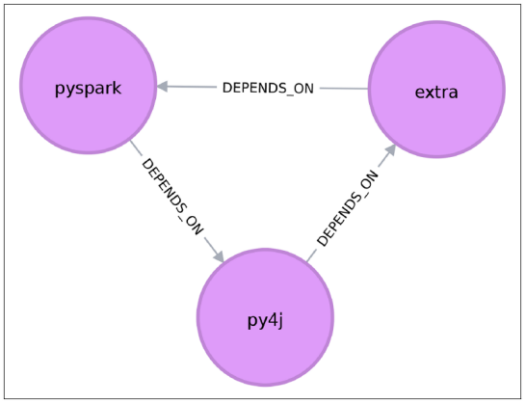
Every node is in its own partition (random each run) but let's create a circular dependency in the graph to make things more interesting. This should mean that we'll end up with some nodes in the same partition.

Community Detection Algorithms

SCC Application – Software Dependency Graph

partition	libraries
1	[pyspark, py4j, extra]
8	[ipykernel]
11	[six]
2	[matplotlib]
5	[jupyter]
14	[numpy]
13	[pandas]
7	[nbconvert]
10	[jpy-core]
9	[jpy-client]
3	[spacy]
15	[python-dateutil]
6	[jpy-console]
0	[pytz]

A circular dependency between pyspark, py4j, and extra. Now, pyspark, py4j, and extra are all part of the same partition, and SCCs helped us find the circular dependency!



Community Detection Algorithms

Connected Components (Union Find)

The Connected Components algorithm (sometimes called Union Find or Weakly Connected Components) finds sets of connected nodes in an undirected graph where each node is reachable from any other node in the same set. It differs from the SCC algorithm because it only needs a path to exist between pairs of nodes in one direction, whereas SCC needs a path to exist in both directions. Bernard A. Galler and Michael J. Fischer first described this algorithm in their 1964 paper, “An Improved Equivalence Algorithm”.

Community Detection Algorithms

Connected Components – Use Cases

- Keeping track of clusters of database records, as part of the deduplication process. Deduplication is an important task in master data management applications; the approach is described in more detail in "An Efficient Domain-Independent Algorithm for Detecting Approximately Duplicate Database Records", by A. Monge and C. Elkan.

- Analyzing citation networks. One study uses Connected Components to work out how well connected a network is, and then to see whether the connectivity remains if "hub" or "authority" nodes are moved from the graph. This use case is explained further in "Characterizing and Mining Citation Graph of Computer Science Literature", a paper by Y. An, J. C. M. Janssen, and E. E. Milios.

Community Detection Algorithms

When to Use Connected Components

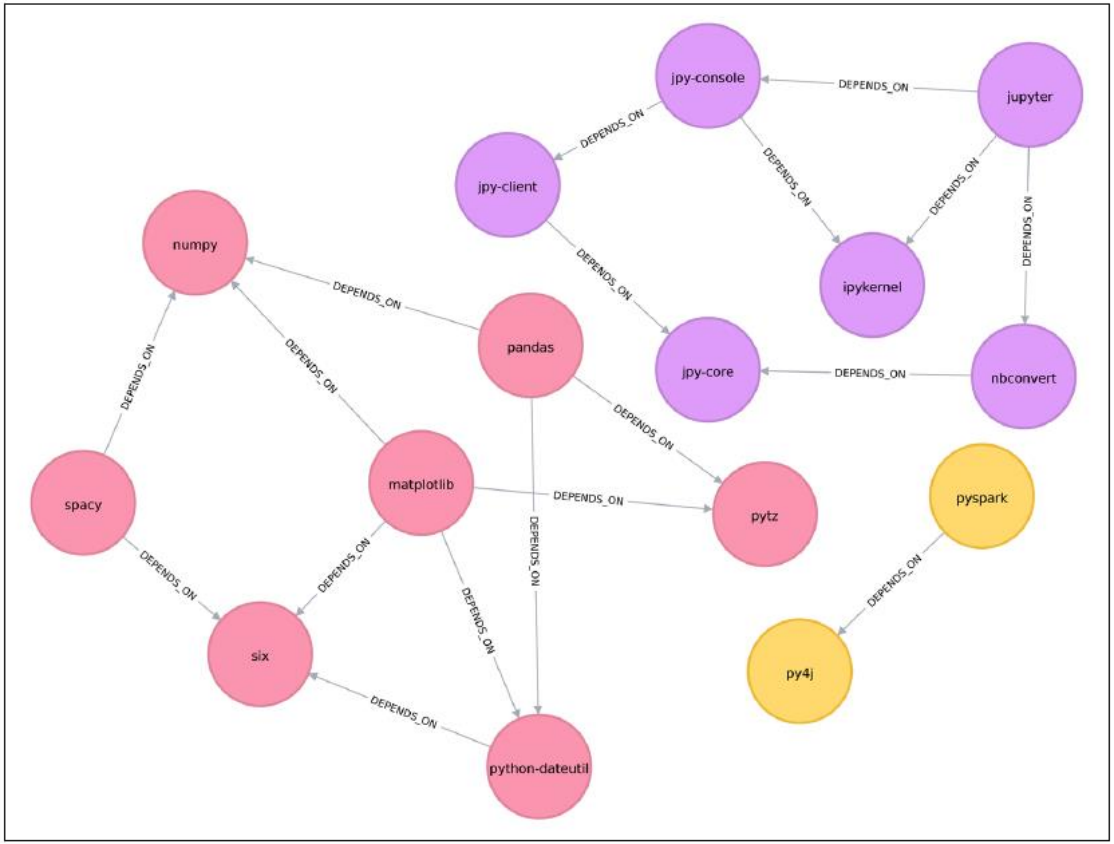
As with SCC, Connected Components is often used early in an analysis to understand a graph's structure. Because it scales efficiently, consider this algorithm for graphs requiring frequent updates. It can quickly show new nodes in common between groups, which is useful for analysis such as fraud detection. Make it a habit to run Connected Components to test whether a graph is connected as a preparatory step for general graph analysis. Performing this quick test can avoid accidentally running algorithms on only one disconnected component of a graph and getting incorrect results.

Community Detection Algorithms

Connected Components – Software Dependency Graph

The results show three clusters of nodes, which can also be seen in Figure (next slide). In this example it's very easy to see that there are three components just by visual inspection. This algorithm shows its value more on larger graphs, where visual inspection isn't possible or is very time-consuming.

componentId	libraries
2	[pytz, matplotlib, spacy, six, pandas, numpy, python-dateutil]
5	[jupyter, jpy-console, nbconvert, ipykernel, jpy-client, jpy-core]
1	[pyspark, py4j]



Community Detection Algorithms - Connected Components in the Software Dependency Graph

Community Detection Algorithms

Deterministic vs Non-Deterministic

Both community detection algorithms that we've covered so far are deterministic: they return the same results each time we run them. Our next two algorithms are examples of nondeterministic algorithms, where we may see different results if we run them multiple times, even on the same data.

Community Detection Algorithms

Label Propagation

The Label Propagation algorithm (LPA) is a fast algorithm for finding communities in a graph. In LPA, nodes select their group based on their direct neighbours. This process is well suited to networks where groupings are less clear and weights can be used to help a node determine which community to place itself within. It also lends itself well to semi-supervised learning because you can seed the process with preassigned, indicative node labels.

Community Detection Algorithms

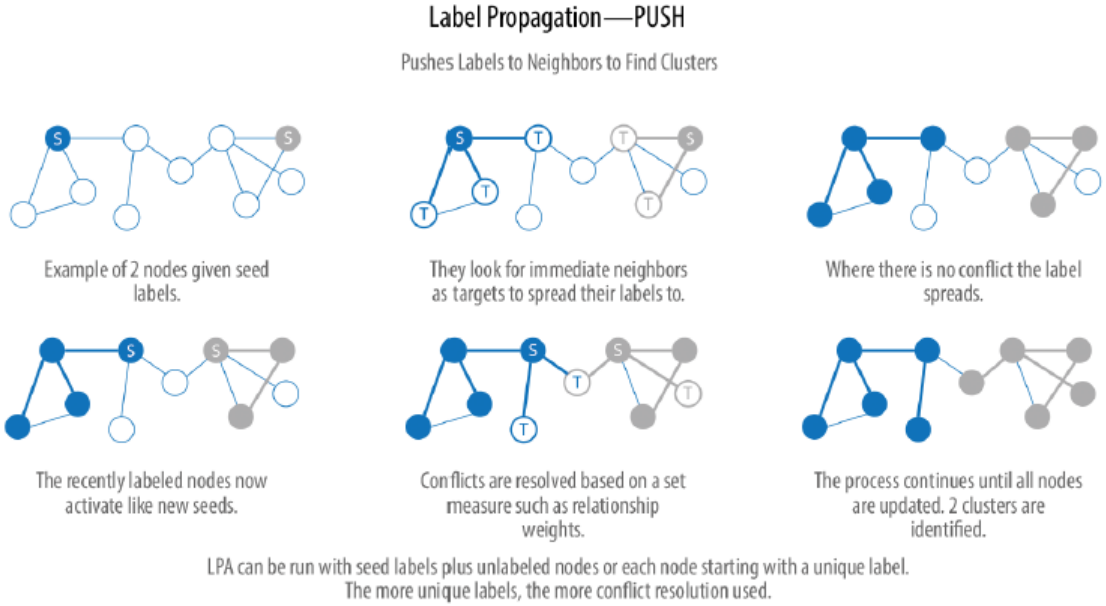
Label Propagation

The intuition behind this algorithm is that a single label can quickly become dominant in a densely connected group of nodes, but it will have trouble crossing a sparsely connected region. Labels get trapped inside a densely connected group of nodes, and nodes that end up with the same label when the algorithm finishes are considered part of the same community. The algorithm resolves overlaps, where nodes are potentially part of multiple clusters, by assigning membership to the label neighbourhood with the highest combined relationship and node weight. LPA is a relatively new algorithm proposed in 2007 by U. N. Raghavan, R. Albert, and S. Kumara, in a paper titled "Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks". The Figure (next slides) depicts two variations of Label Propagation, a simple **push** method and the more typical **pull** method that relies on relationship weights. The pull method lends itself well to parallelization.

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

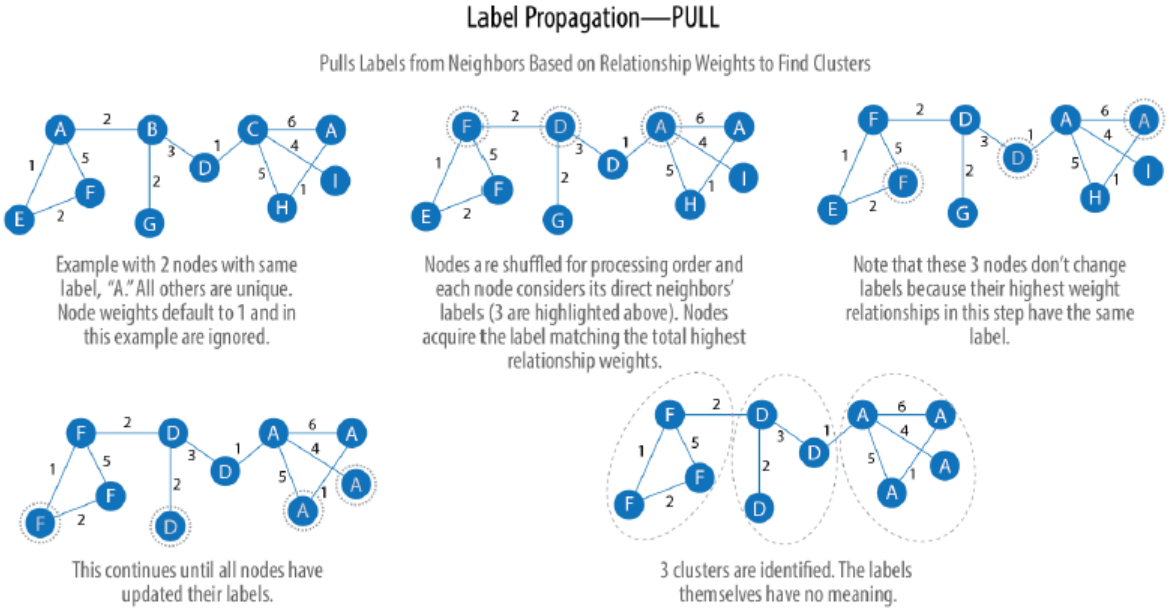
Community Detection Algorithms

Label Propagation – Push Method



Community Detection Algorithms

Label Propagation – Pull Method



Community Detection Algorithms

Label Propagation – Pull Method - Steps

The steps often used for the Label Propagation pull method are:

1. Every node is initialized with a unique label (an identifier), and optionally preliminary “seed” labels can be used.
2. These labels propagate through the network.
3. At every propagation iteration, each node updates its label to match the one with the maximum weight, which is calculated based on the weights of neighbour nodes *and* their relationships. Ties are broken uniformly and randomly.
4. LPA reaches convergence when each node has the majority label of its neighbours.

Graph Algorithms Mark Needham and Amy E. Hodler, O’Reilly 2019

Community Detection Algorithms

Label Propagation – Pull Method - Steps

As labels propagate, densely connected groups of nodes quickly reach a consensus on a unique label. At the end of the propagation, only a few labels will remain, and nodes that have the same label belong to the same community.

Community Detection Algorithms – Label Propagation

Semi-Supervised Learning and Seed Labels

In contrast to other algorithms, Label Propagation can return different community structures when run multiple times on the same graph. The order in which LPA evaluates nodes can have an influence on the final communities it returns.

The range of solutions is narrowed when some nodes are given preliminary labels (i.e., seed labels), while others are unlabelled. Unlabelled nodes are more likely to adopt the preliminary labels.

Community Detection Algorithms – Label Propagation

Semi-Supervised Learning and Seed Labels

This use of Label Propagation can be considered a **semi-supervised learning** method to find communities. Semi-supervised learning is a class of machine learning tasks and techniques that operate on a small amount of labelled data, along with a larger amount of unlabelled data. We can also run the algorithm repeatedly on graphs as they evolve.

Community Detection Algorithms – Label Propagation

Semi-Supervised Learning and Seed Labels

Finally, LPA sometimes doesn't converge on a single solution. In this situation, our community results will continually flip between a few remarkably similar communities and the algorithm would never complete. Seed labels help guide it toward a solution.

Neo4j use a set maximum number of iterations to avoid never-ending execution. You should test the iteration setting for your data to balance accuracy and execution time.

Community Detection Algorithms

When to Use Label Propagation

Use Label Propagation in large-scale networks for initial community detection, especially when weights are available. This algorithm can be parallelized and is therefore extremely fast at graph partitioning.

Community Detection Algorithms

Label Propagation – Use Cases

- Assigning polarity of tweets as a part of semantic analysis. In this scenario, positive and negative seed labels from a classifier are used in combination with the Twitter follower graph. For more information, see "Twitter Polarity Classification with Label Propagation over Lexical Links and the Follower Graph", by M. Speriosu et al.

- Finding potentially dangerous combinations of possible co-prescribed drugs, based on the chemical similarity and side effect profiles. See "Label Propagation Prediction of Drug–Drug Interactions Based on Clinical Side Effects", a paper by P. Zhang et al.

Community Detection Algorithms

Label Propagation – Use Cases

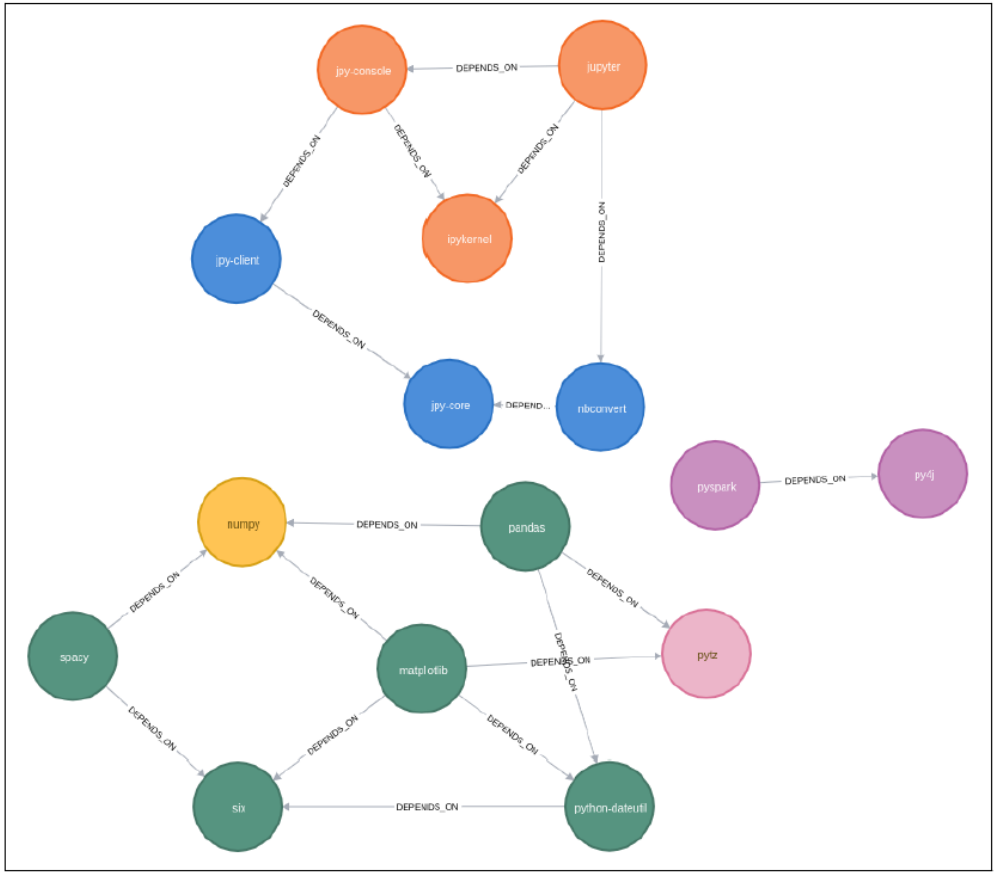
- Inferring dialogue features and user intention for a machine learning model. For more information, see “Feature Inference Based on Label Propagation on Wikidata Graph for DST”, a paper by Y. Murase et al.

Community Detection Algorithms

Label Propagation – Software Dependency Graph

Compared to Connected Components, we have more clusters of libraries in this example. LPA is less strict than Connected Components with respect to how it determines clusters. Two neighbours (directly connected nodes) may be found to be in different clusters using Label Propagation. However, using Connected Components a node would always be in the same cluster as its neighbours because that algorithm bases grouping strictly on relationships.

In our example, the most obvious difference is that the Jupyter libraries have been split into two communities—one containing the core parts of the library and the other the client-facing tools.

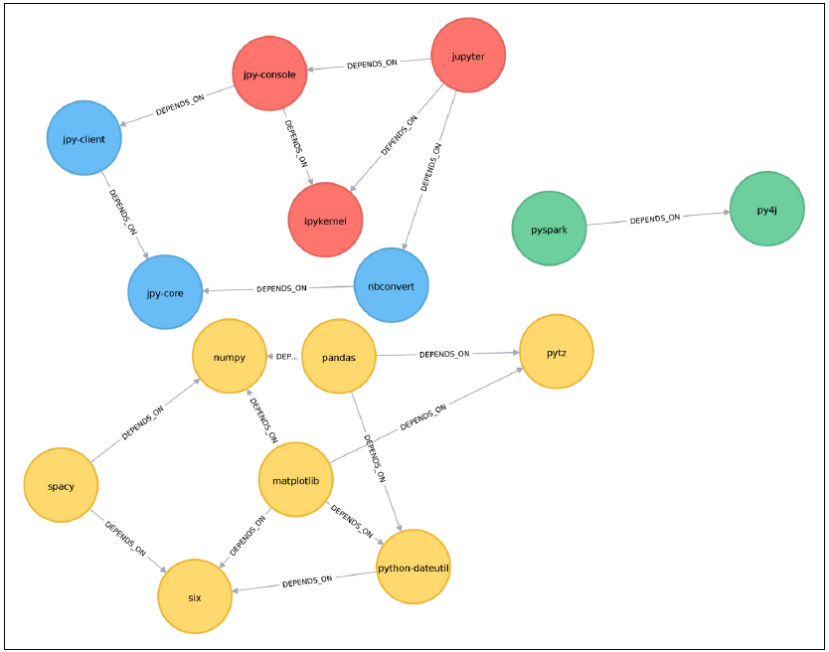


Community Detection Algorithms – Label Propagation in the Software Dependency Graph

graph	libraries
32	["six", "pandas", "python-dateutil", "matplotlib", "spacy"]
0	["ipykernel", "jupyter", "jpy-console"]
2	["jpy-client", "jpy-core", "nbconvert"]
908	["pyspark", "py4j"]
34	["numpy"]
36	["pytz"]

Community Detection Algorithms

Label Propagation – Software Dependency Graph



We can also run the algorithm assuming that the graph is undirected, which means that nodes will try to adopt labels from the libraries they depend on as well as ones that depend on them. The **number of clusters is reduced** from six to three/four, and **all the nodes in the matplotlib part of the graph are now grouped together**. This can be seen more clearly in sample output (left).

Community Detection Algorithms

Label Propagation – Software Dependency Graph

Although the results of running Label Propagation on this data are similar for undirected and directed calculation, on complicated graphs you will see more significant differences. This is because ignoring direction causes nodes to try and adopt more labels, regardless of the relationship source.

Community Detection Algorithms

Louvain Modularity

The **Louvain Modularity** algorithm finds clusters by comparing community density as it assigns nodes to different groups. You can think of this as a “what if ” analysis to try various groupings with the goal of reaching a global optimum.

Proposed in 2008, the Louvain algorithm is one of the fastest modularity-based algorithms. As well as detecting communities, it also reveals a hierarchy of communities at different scales. This is useful for understanding the structure of a network at different levels of granularity.

Louvain quantifies how well a node is assigned to a group by looking at the density of connections within a cluster in comparison to an average or random sample. This measure of community assignment is called **modularity**.

Community Detection Algorithms

Louvain – Quality based grouping via Modularity

Modularity is a technique for uncovering communities by partitioning a graph into more coarse-grained modules (or clusters) and then measuring the strength of the groupings. As opposed to just looking at the concentration of connections within a cluster, this method compares relationship densities in given clusters to densities between clusters. The measure of the quality of those groupings is called **modularity**.

Community Detection Algorithms

Louvain – Quality based grouping via Modularity

Modularity algorithms optimize communities locally and then globally, using multiple iterations to test different groupings and increasing coarseness. This strategy identifies community hierarchies and provides a broad understanding of the overall structure.

However, all modularity algorithms suffer from two drawbacks:

- They merge smaller communities into larger ones.
- A plateau can occur where several partition options are present with similar modularity, forming local maxima and preventing progress.

For more information, see the paper "The Performance of Modularity Maximization in Practical Contexts", by B. H. Good, Y.-A. de Montjoye, and A. Clauset.

Community Detection Algorithms

Louvain – Quality based grouping via Modularity

Initially the Louvain Modularity algorithm optimizes modularity locally on all nodes, which finds small communities; then each small community is grouped into a larger conglomerate node and the first step is repeated until we reach a global optimum.

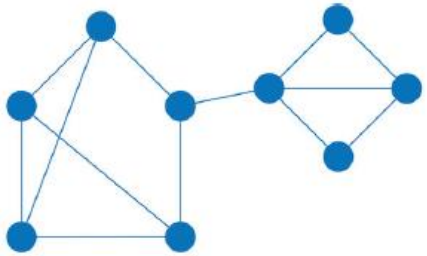
Community Detection Algorithms

Louvain – Calculating Modularity

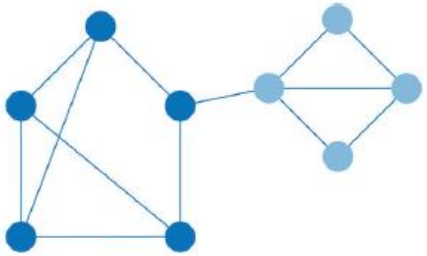
A simple calculation of modularity is based on the fraction of the relationships within the given groups minus the expected fraction if relationships were distributed at random between all nodes. The value is always between 1 and –1, with positive values indicating more relationship density than you'd expect by chance and negative values indicating less density. Figure (next slide) illustrates several different modularity scores based on node groupings.
The formula for the modularity of a group is:

$M = \sum_{c=1}^{n_c} [\frac{L_c}{L} - (\frac{k_c}{2L})^2]$ where:

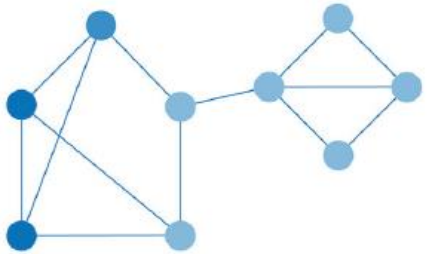
L is the number of relationships in the entire group.
L_c is the number of relationships in a partition.
k_c is the total degree of nodes in a partition.



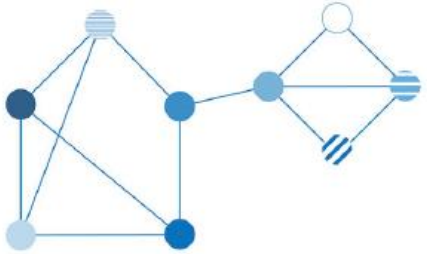
Random Baseline
(Single Community)
M = 0.0



Optimal Partition
M = 0.41



Suboptimal Partition
M = 0.22



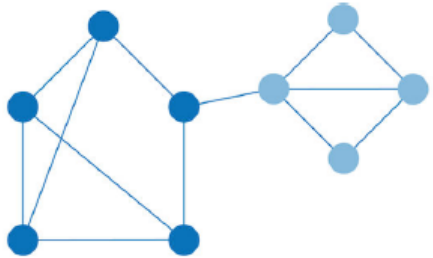
Negative Modularity
M = - 0.12

Louvain Modularity
Four modularity scores based on different partitioning choices

Community Detection Algorithms

Louvain – Calculating Modularity

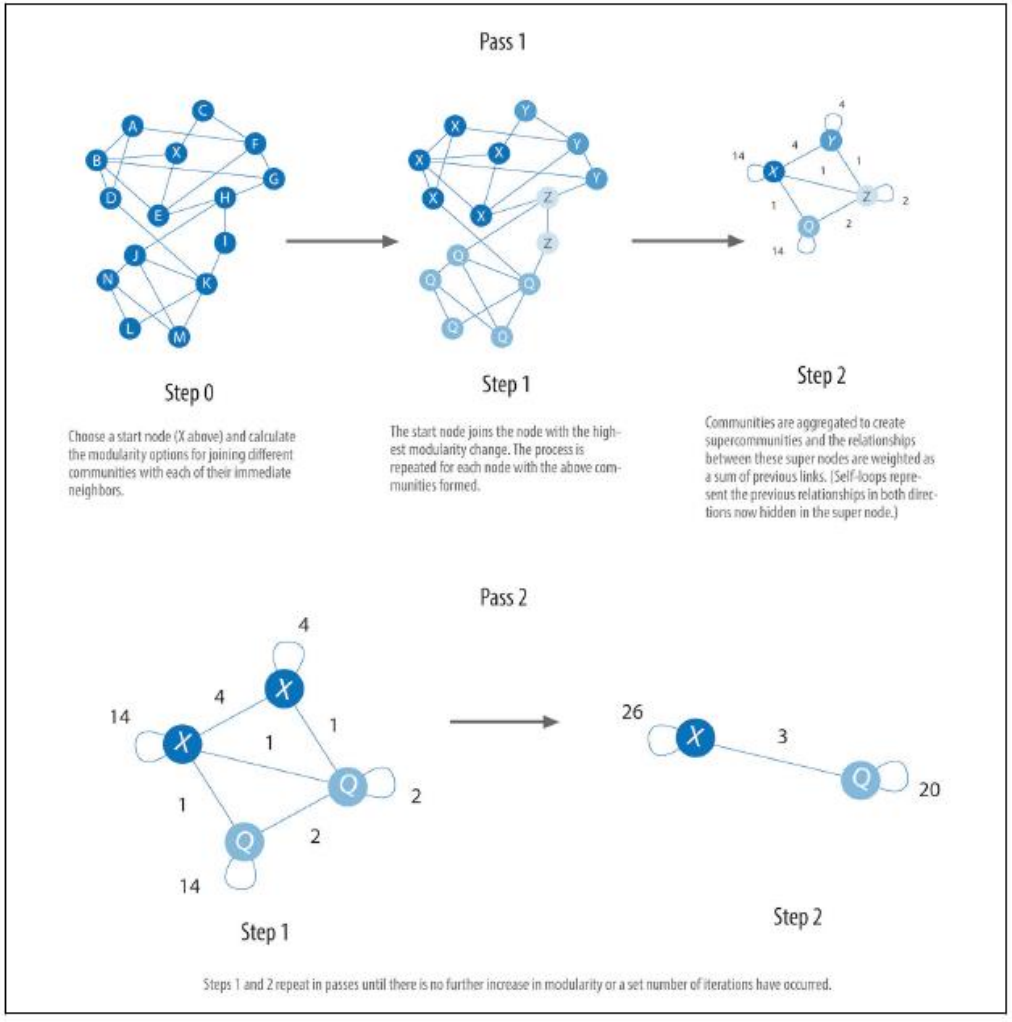
$M = \sum_{c=1}^{n_c}[\frac{L_c}{L} - (\frac{k_c}{2L})^2]$ where:



L is the number of relationships in the entire group.
L_c is the number of relationships in a partition.
k_c is the total degree of nodes in a partition.

Calculation for the optimal partition at the top Figure (previous slide) is as follows:

- The dark partition is $\left(\frac{7}{13} - \left(\frac{15}{2(13)}\right)^2\right) = 0.205$
- The light partition is $\left(\frac{5}{13} - \left(\frac{11}{2(13)}\right)^2\right) = 0.206$
- These are added together for $\mathrm{M} = 0.205 + 0.206 = 0.41$



The Louvain algorithm consists of repeated application of two steps, as illustrated in the figure (right).

Community Detection Algorithms

Louvain algorithm steps

The Louvain algorithm's steps include:

1. A "greedy" assignment of nodes to communities, favouring local optimizations of modularity.
2. The definition of a more coarse-grained network based on the communities found in the first step. This coarse-grained network will be used in the next iteration of the algorithm.

These two steps are repeated until no further modularity-increasing reassignments of communities are possible.

Community Detection Algorithms

Louvain algorithm steps

$$Q = \frac{1}{2m}\sum_{u,v}\left[A_{uv} - \frac{k_u k_v}{2m}\right]\delta(c_u, c_v)$$

where:

- u and v are nodes.
- m is the total relationship weight across the entire graph ($2m$ is a common normalization value in modularity formulas).
- $A_{uv} - \frac{k_u k_v}{2m}$ is the strength of the relationship between u and v compared to what we would expect with a random assignment (tends toward averages) of those nodes in the network.
  - A_{uv} is the weight of the relationship between u and v.
  - k_u is the sum of relationship weights for u.
  - k_v is the sum of relationship weights for v.
- $\delta(c_u, c_v)$ is equal to 1 if u and v are assigned to the same community, and 0 if they are not.

Part of the first optimization step is evaluating the modularity of a group. Louvain uses the formula (left). Another part of that first step evaluates the change in modularity if a node is moved to another group. Louvain uses a more complicated variation of this formula and then determines the best group assignment.

Community Detection Algorithms

When to Use Louvain

Use Louvain Modularity to find communities in vast networks. This algorithm applies a heuristic, as opposed to exact, modularity, which is computationally expensive. Louvain can therefore be used on large graphs on which standard modularity algorithms may struggle.

Louvain is also very helpful for evaluating the structure of complex networks, in particular uncovering many levels of hierarchies–such as what you might find in a criminal organization. The algorithm can provide results where you can zoom in on different levels of granularity and find subcommunities within subcommunities within subcommunities.

Community Detection Algorithms

Louvain – Use Cases

Example use cases include:

- Detecting cyberattacks. The Louvain algorithm was used in a 2016 study by S. V. Shanbhaq of fast community detection in large-scale cybernetworks for cybersecurity applications. Once these communities have been detected they can be used to detect cyberattacks.
- Extracting topics from online social platforms, like Twitter and YouTube, based on the co-occurence of terms in documents as part of the topic modelling process. This approach is described in a paper by G. S. Kido, R. A. Igawa, and S. Barbon Jr., "Topic Modeling Based on Louvain Method in Online Social Networks".
- Finding hierarchical community structures within the brain's functional network, as described in "Hierarchical Modularity in Human Brain Functional Networks" by D. Meunier et al.

Community Detection Algorithms

Louvain – Application Guidelines

Modularity optimization algorithms, including Louvain, suffer from two issues. First, the algorithms can overlook small communities within large networks. You can overcome this problem by reviewing the intermediate consolidation steps. Second, in large graphs with overlapping communities, modularity optimizers may not correctly determine the global maxima. In the latter case, it is recommended to use any modularity algorithm as a guide for gross estimation but not complete accuracy.

Community Detection Algorithms

Louvain – Software Dependency Graph

libraries	communityId	intermediateCommunityIds
"ipykernel"	0	[0, 0]
"jpy-client"	0	[2, 0]
"jpy-core"	0	[2, 0]
"six"	5	[3, 5]
"pandas"	5	[7, 5]
"numpy"	5	[5, 5]
"python-dateutil"	5	[3, 5]
"pytz"	5	[7, 5]
"pyspark"	11	[11, 11]
"matplotlib"	5	[3, 5]
"spacy"	5	[5, 5]
"py4j"	11	[11, 11]
"jupyter"	0	[2, 0]
"jpy-console"	0	[2, 0]
"nbconvert"	0	[2, 0]

The intermediateCommunityIds column describes the community that nodes fall into at two levels. The last value in the array is the final community and the other one is an intermediate community. The numbers assigned to the intermediate and final communities are simply labels with no measurable meaning. Treat these as labels that indicate which community nodes belong to such as "belongs to a community labeled 0", "a community labeled 4", and so forth. For example, matplotlib has a result of [3,5]. This means that matplotlib's final community is labelled 5 and its intermediate community is labelled 3.

Community Detection Algorithms - Louvain

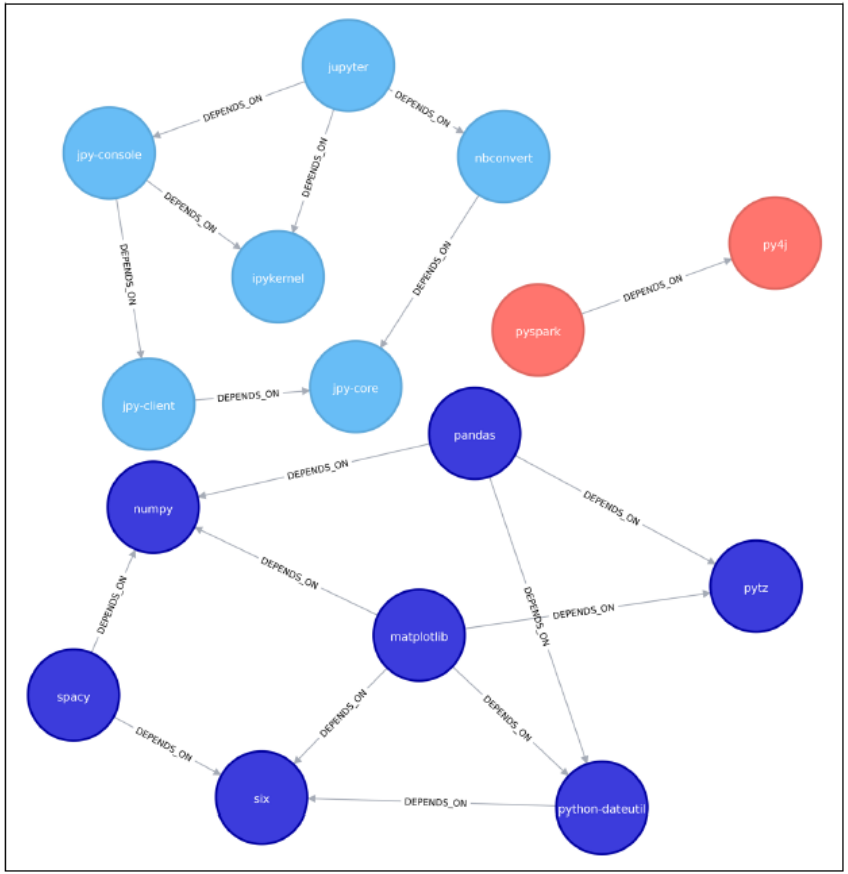
Software Dependency Graph – Final Clustering

community	libraries
5	["six", "pandas", "numpy", "python-dateutil", "pytz", "matplotlib", "spacy"]
0	["ipykernel", "jpy-client", "jpy-core", "jupyter", "jpy-console", "nbconvert"]
11	["pyspark", "py4j"]

This clustering is the same as we saw with the connected components algorithm.

matplotlib is in a community with pytz, spacy, six, pandas, numpy, and pythondateutil.

We can see this more clearly in Figure (next slide).



Community Detection
Louvain Modularity
Software Dependency
Graph Final Clustering

Community Detection Algorithms - Louvain

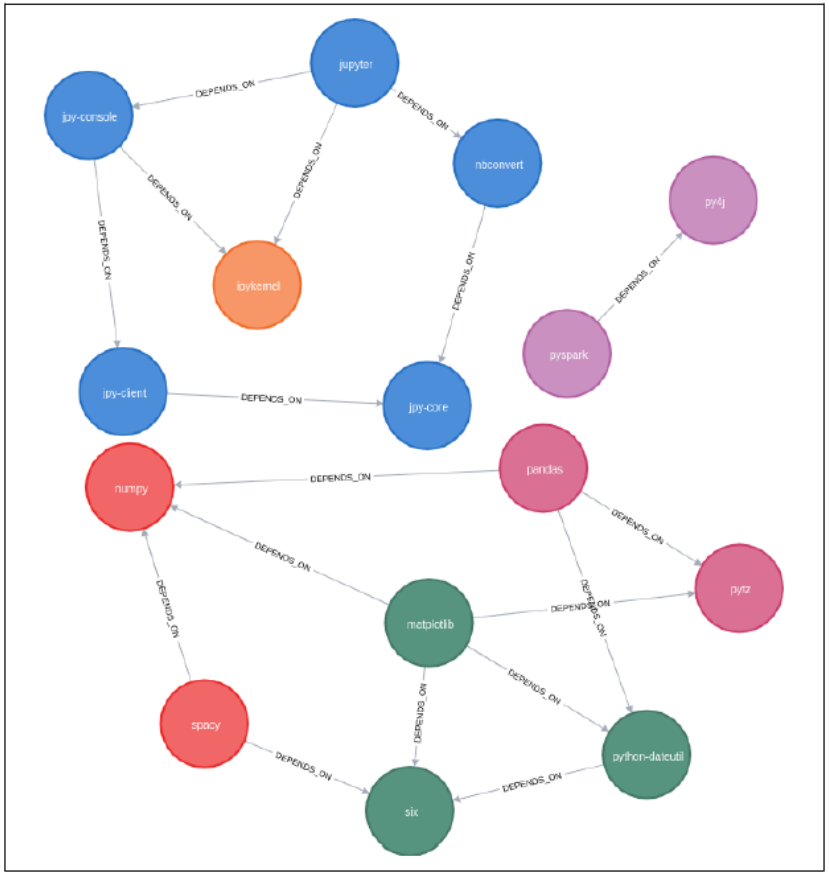
Software Dependency Graph – Intermediate Clustering

An additional feature of the Louvain algorithm is that we can see the intermediate clustering as well. The following query will show us finer-grained clusters than shown in the final layer (sample output below). The libraries in the matplotlib community have now broken down into three smaller communities:

- matplotlib, python-dateutil, and six
- pandas and pytz
- numpy and spacy

We can see this breakdown visually in Figure (next slide).

community	libraries
2	["jpy-client", "jpy-core", "jupyter", "jpy-console", "nbconvert"]
3	["six", "python-dateutil", "matplotlib"]
7	["pandas", "pytz"]
5	["numpy", "spacy"]
11	["pyspark", "py4j"]
0	["ipykernel"]



Community Detection
Louvain Modularity
Software Dependency Graph
Intermediate Clustering

Community Detection Algorithms - Louvain

Software Dependency Graph – Intermediate Clustering

Although this graph only showed two layers of hierarchy, if we ran this algorithm on a larger graph, we would see a more complex hierarchy. The intermediate clusters that Louvain reveals can be very useful for detecting fine-grained communities that may not be detected by other community detection algorithms.

Community Detection Algorithms

Validating Communities

Community detection algorithms generally have the same goal: to identify groups. However, because different algorithms begin with different assumptions, they may uncover different communities. This makes choosing the right algorithm for a particular problem more challenging and a bit of an exploration.

Most community detection algorithms do reasonably well when relationship density is high within groups compared to their surroundings, but real-world networks are often less distinct. We can validate the accuracy of the communities found by comparing our results to a benchmark based on data with known communities.

Community Detection Algorithms

Validating Communities - Benchmarks

Two of the best-known benchmarks are the **Girvan-Newman (GN)** and **Lancichinetti–Fortunato–Radicchi (LFR)** algorithms. The reference networks that these algorithms generate are quite different: GN generates a random network which is more homogeneous, whereas LFR creates a more heterogeneous graph in which node degrees and community size are distributed according to a power law.

Since the accuracy of our testing depends on the benchmark used, it's important to match our benchmark to our dataset. As much as possible, look for similar densities, relationship distributions, community definitions, and related domains.

Community Detection Algorithms

Summary

Community detection algorithms are useful for understanding the way that nodes are grouped together in a graph.

We covered the Triangle Count and Clustering Coefficient algorithms. We then moved on to two deterministic community detection algorithms: Strongly Connected Components and Connected Components. These algorithms have strict definitions of what constitutes a community and are very useful for getting a feel for the graph structure early in the graph analytics pipeline.

Finally, we explored Label Propagation and Louvain, two nondeterministic algorithms which are better able to detect finer-grained communities. Louvain also reveals a hierarchy of communities at different scales.

Centrality Algorithms

Introduction

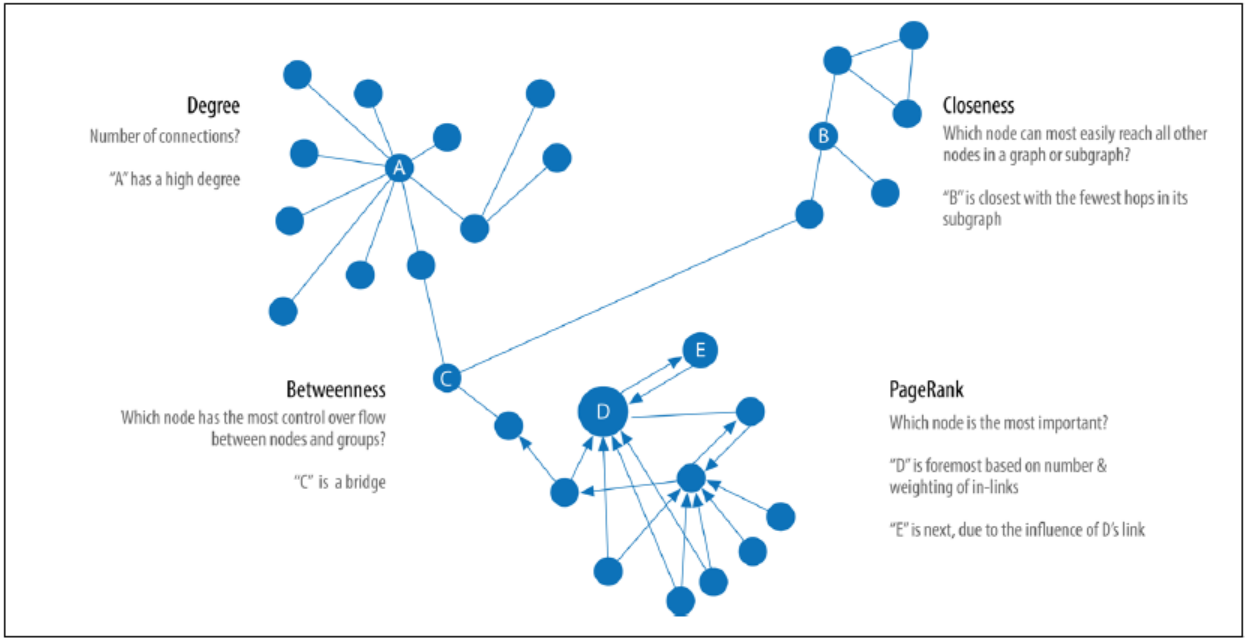
Centrality algorithms are used to understand the roles of particular nodes in a graph and their impact on that network. They're useful because they identify the most important nodes and help us understand group dynamics such as credibility, accessibility, the speed at which things spread, and bridges between groups. Although many of these algorithms were invented for social network analysis, they have since found uses in a variety of industries and fields.

- **Degree Centrality** as a baseline metric of connectedness
- **Closeness Centrality** for measuring how central a node is to the group, including two variations for disconnected groups
- **Betweenness Centrality** for finding control points, including an alternative for approximation
- **PageRank** for understanding the overall influence, including a popular option for personalization

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Centrality Algorithms

Types of Questions They Answer



Centrality Algorithms

Types of Questions They Answer

Algorithm type	What it does	Example use	Spark example	Neo4j example
Degree Centrality	Measures the number of relationships a node has	Estimating a person's popularity by looking at their in-degree and using their out-degree to estimate gregariousness	Yes	No
Closeness Centrality Variations: Wasserman and Faust, Harmonic Centrality	Calculates which nodes have the shortest paths to all other nodes	Finding the optimal location of new public services for maximum accessibility	Yes	Yes
Betweenness Centrality Variation: Randomized-Approximate Brandes	Measures the number of shortest paths that pass through a node	Improving drug targeting by finding the control genes for specific diseases	No	Yes
PageRank Variation: Personalized PageRank	Estimates a current node's importance from its linked neighbors and their neighbors (popularized by Google)	Finding the most influential features for extraction in machine learning and ranking text for entity relevance in natural language processing.	Yes	Yes

Centrality Algorithms

Example Graph Data – Twitter Social Graph

Centrality algorithms are relevant to all graphs, but social networks provide a very relatable way to think about dynamic influence and the flow of information. The examples in this module are run against a small Twitter-like graph. You can load the nodes and relationships files using the accompanying CYPHER script “import centrality.cypher.”

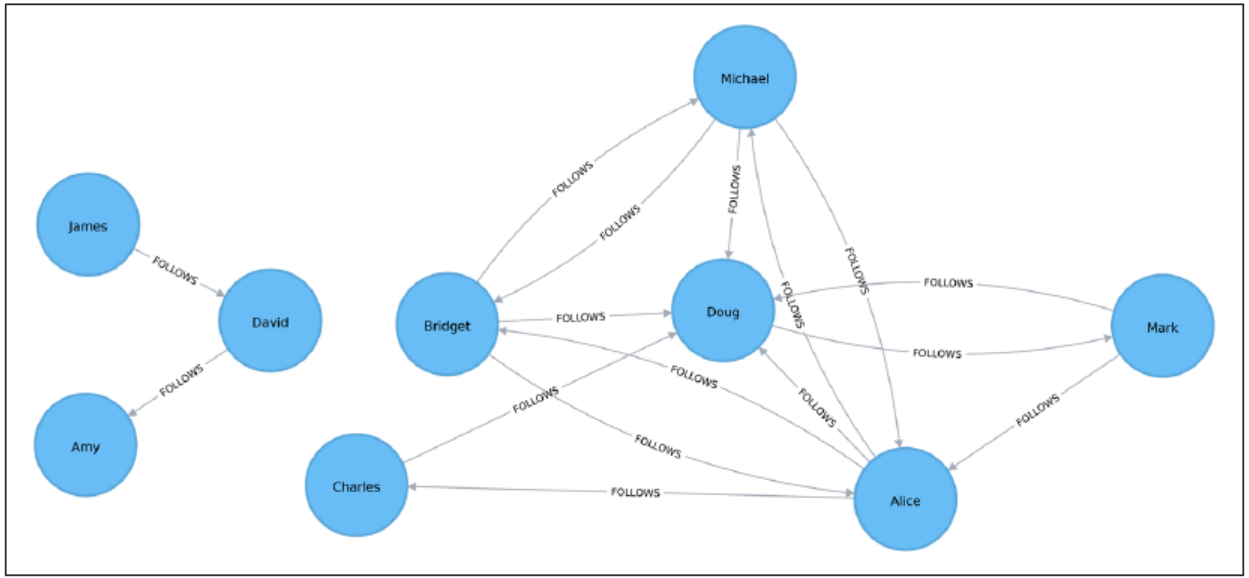
id
Alice
Bridget
Charles
Doug
Mark
Michael
David
Amy
James

src	dst	relationship
Alice	Bridget	FOLLOWS
Alice	Charles	FOLLOWS
Mark	Doug	FOLLOWS
Bridget	Michael	FOLLOWS
Doug	Mark	FOLLOWS
Michael	Alice	FOLLOWS
Alice	Michael	FOLLOWS
Bridget	Alice	FOLLOWS
Michael	Bridget	FOLLOWS

src	dst	relationship
Charles	Doug	FOLLOWS
Bridget	Doug	FOLLOWS
Michael	Doug	FOLLOWS
Alice	Doug	FOLLOWS
Mark	Alice	FOLLOWS
David	Amy	FOLLOWS
James	David	FOLLOWS

Centrality Algorithms

Twitter Social Graph Model



Centrality Algorithms

Degree Centrality

Degree Centrality is the simplest of the algorithms. It counts the number of incoming and outgoing relationships from a node and is used to find popular nodes in a graph. Degree Centrality was proposed by Linton C. Freeman in his 1979 paper "Centrality in Social Networks: Conceptual Clarification".

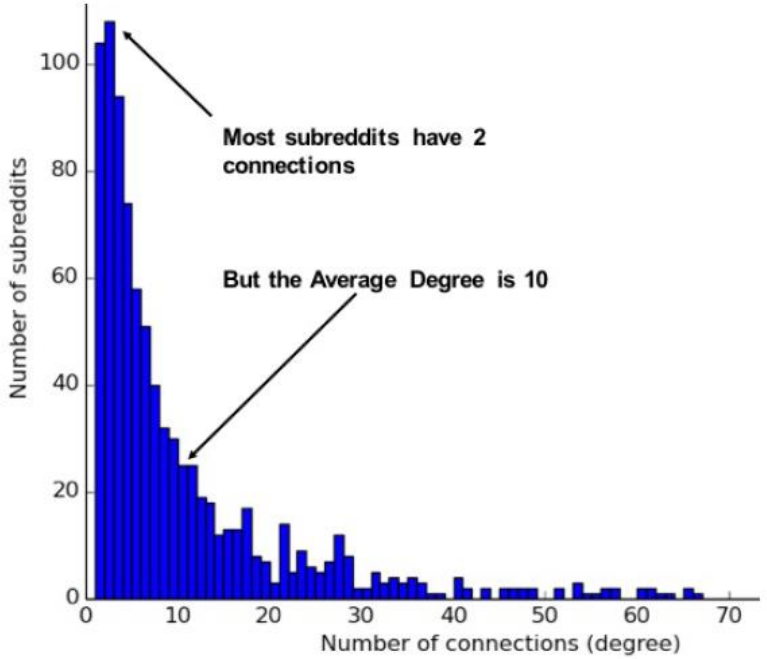
Reach

Understanding the reach of a node is a fair measure of importance. How many other nodes can it touch right now? The **degree** of a node is the number of direct relationships it has, calculated for in-degree and out-degree. You can think of this as the immediate reach of node. For example, a person with a high degree in an active social network would have a lot of immediate contacts and be more likely to catch a virus circulating in their network.

Centrality Algorithms

Average Degree of a Network

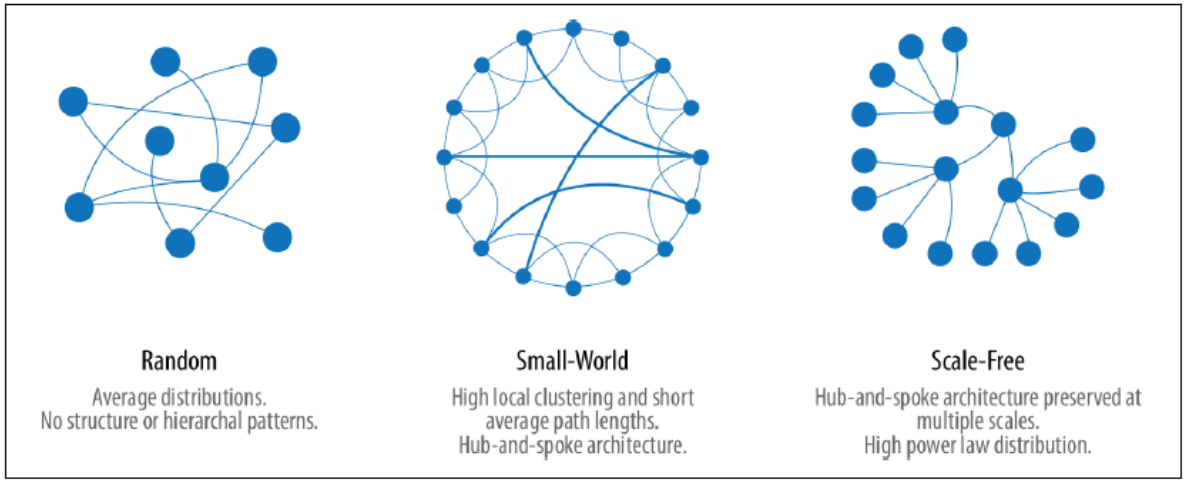
The **average degree** of a network is simply the total number of relationships divided by the total number of nodes; it can be heavily skewed by high degree nodes. The **degree distribution** is the probability that a randomly selected node will have a certain number of relationships. The actual distribution of connections among subreddit topics (Figure). If you simply took the average, you'd assume most topics have 10 connections, whereas in fact most topics only have 2 connections.



Centrality Algorithms

Average Degree of a Network

These measures are used to categorize network types such as the **scale-free** or **small-world** networks. They also provide a quick measure to help estimate the potential for things to spread or ripple throughout a network.



Centrality Algorithms

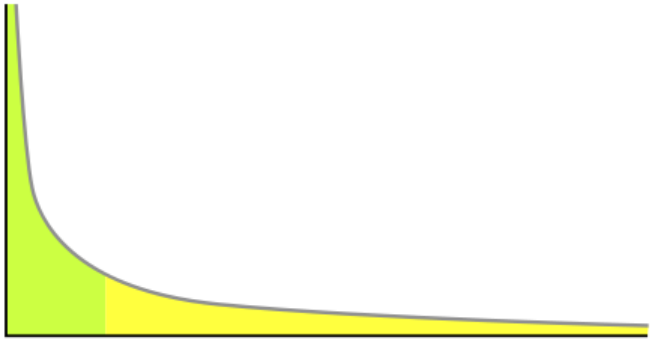
Average Degree of a Network – Small World

A **small-world network** is a type of mathematical graph in which most nodes are not neighbours of one another, but the neighbours of any given node are likely to be neighbours of each other and most nodes can be reached from every other node by a **small** number of hops or steps.

Centrality Algorithms

Average Degree of a Network – Scale-Free

A **scale-free network** is a network whose degree distribution follows a power law (one quantity varies as a power of another) at least asymptotically.



Centrality Algorithms

When to Use Degree Centrality

Use Degree Centrality if you're attempting to analyse influence by looking at the number of incoming and outgoing relationships or find the "popularity" of individual nodes. It works well when you're concerned with immediate connectedness or near-term probabilities. However, Degree Centrality is also applied to global analysis when you want to evaluate the minimum degree, maximum degree, mean degree, and standard deviation across the entire graph.

Centrality Algorithms

Degree Centrality - Example Use Cases

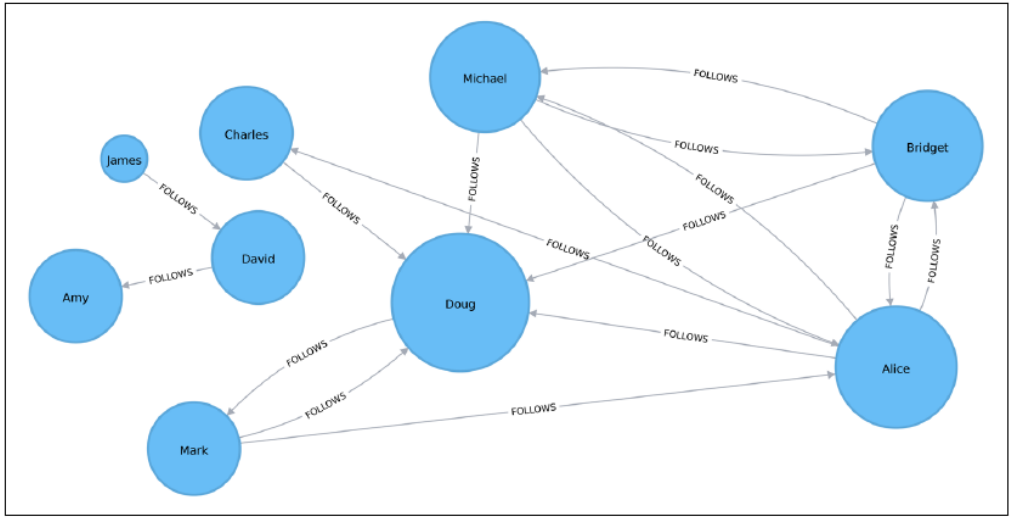
- Identifying powerful individuals though their relationships, such as connections of people in a social network. For example, in BrandWatch's "Most Influential Men and Women on Twitter 2017", the top 5 people in each category have over 40 million followers each. https://www.brandwatch.com/blog/react-influential-men-and-women-2017/

- Separating fraudsters from legitimate users of an online auction site. The weighted centrality of fraudsters tends to be significantly higher due to collusion aimed at artificially increasing prices. Read more in the paper by P. Bangcharoensap et al., "Two Step Graph-Based Semi-Supervised Learning for Online Auction Fraud Detection".

Centrality Algorithms

Degree Centrality - Example Use Cases

Doug is the most popular user in our Twitter graph, with five followers (in-links). All other users in that part of the graph follow him and he only follows one person back. In the real Twitter network, celebrities have high follower counts but tend to follow few people. We could therefore consider Doug a celebrity!



Centrality Algorithms

Closeness Centrality

Closeness Centrality is a way of detecting nodes that are able to spread information efficiently through a subgraph. The measure of a node's centrality is its average farness (inverse distance) to all other nodes. Nodes with a high closeness score have the shortest distances from all other nodes. For each node, the Closeness Centrality algorithm calculates the sum of its distances to all other nodes, based on calculating the shortest paths between all pairs of nodes. The resulting sum is then *inverted* to determine the closeness centrality score for that node.

Centrality Algorithms

Closeness Centrality

The closeness centrality of a node is calculated using the formula:

$$C(u) = \frac{1}{\sum_{v=1}^{n-1} d(u,v)}$$

where u is a node; n is the number of nodes in the graph; $d(u,v)$ is the shortest-path distance between another node v and u. It is more common to normalize this score so that it represents the average length of the shortest paths rather than their sum. This adjustment allows comparisons of the closeness centrality of nodes of graphs of different sizes. The formula for normalized closeness centrality is as follows:

$$C_{norm}(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(u,v)}$$

Centrality Algorithms

When to Use Closeness Centrality?

Apply Closeness Centrality when you need to know which nodes disseminate things the fastest. Using weighted relationships can be especially helpful in evaluating interaction speeds in communication and behavioural analyses.

Centrality Algorithms

Closeness Centrality – Example Use Cases

- Uncovering individuals in very favourable positions to control and acquire vital information and resources within an organization. One such study is "Mapping Networks of Terrorist Cells", by V. E. Krebs.

- As a heuristic for estimating arrival time in telecommunications and package delivery, where content flows through the shortest paths to a predefined target. It is also used to shed light on propagation through all shortest paths simultaneously, such as infections spreading through a local community. Find more details in "Centrality and Network Flow", by S. P. Borgatti.

- Evaluating the importance of words in a document, based on a graph-based key phrase extraction process. This process is described by F. Boudin in "A Comparison of Centrality Measures for Graph-Based Keyphrase Extraction".

Centrality Algorithms

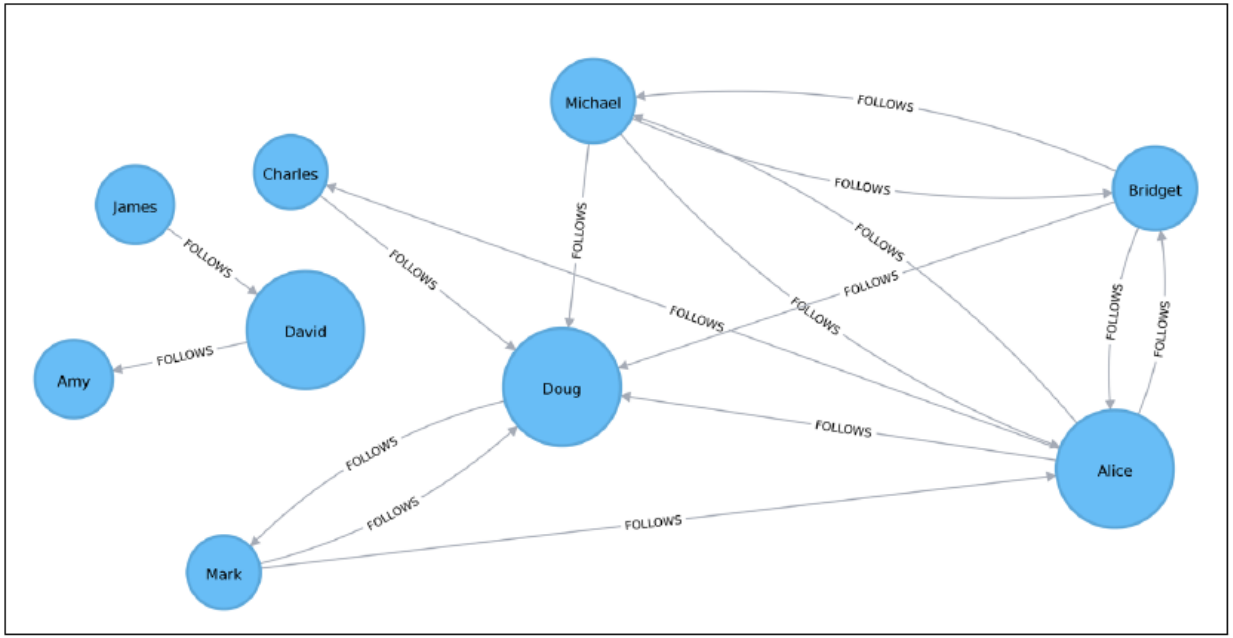
Closeness Centrality – Connected Graphs

Closeness Centrality works best on connected graphs. When the original formula is applied to an unconnected graph, we end up with an infinite distance between two nodes where there is no path between them and an infinite closeness centrality score when we sum up all the distances from that node. To avoid this issue, a variation on the original formula is available. Figure (next slide) illustrates that even though David has only a few connections, within his group of friends that's significant. In other words, this score represents the closeness of each user to others within their subgraph but not the entire graph. Neo4j's implementation of Closeness Centrality uses the formula: $C(u) = \frac{1}{\sum_{v=1}^{n-1} d(u,v)}$

where u is a node; $\boldsymbol{n}$ **is number of nodes in the same component (subgraph or group) as** $\boldsymbol{u}$, $d(u,v)$ is the shortest-path distance between another node v and u.

Centrality Algorithms

Closeness Centrality – Connected Graphs



Centrality Algorithms

Closeness Centrality – Variations

The Closeness Centrality score represents closeness to others within their subgraph but not the entire graph. In the strict interpretation of the Closeness Centrality algorithm, all the nodes in our graph would have a score of ∞ because every node has at least one other node that it's unable to reach. However, it's usually more useful to implement the score per component. Ideally, we'd like to get an indication of closeness across the whole graph. **Wasserman Faust** and **Harmonic Centrality** are two such variations.

Centrality Algorithms

Closeness Centrality Variations – Wasserman Faust

Stanley Wasserman and Katherine Faust came up with an improved formula for calculating closeness for graphs with multiple subgraphs without connections between those groups. Details on their formula are in their book, *Social Network Analysis: Methods and Applications*. The result of this formula is a ratio of the fraction of nodes in the group that are reachable to the average distance from the reachable nodes. The formula is as follows:

$$C_{WF}(u) = \frac{n-1}{N-1}\frac{n-1}{\sum_{v=1}^{n-1} d(u,v)}$$

where u is a node; N is the total node count, n is the number of nodes in the same component as u, $d(u,v)$ is the shortest-path distance between another node v and u. (Neo4j parameter **improved:true**)

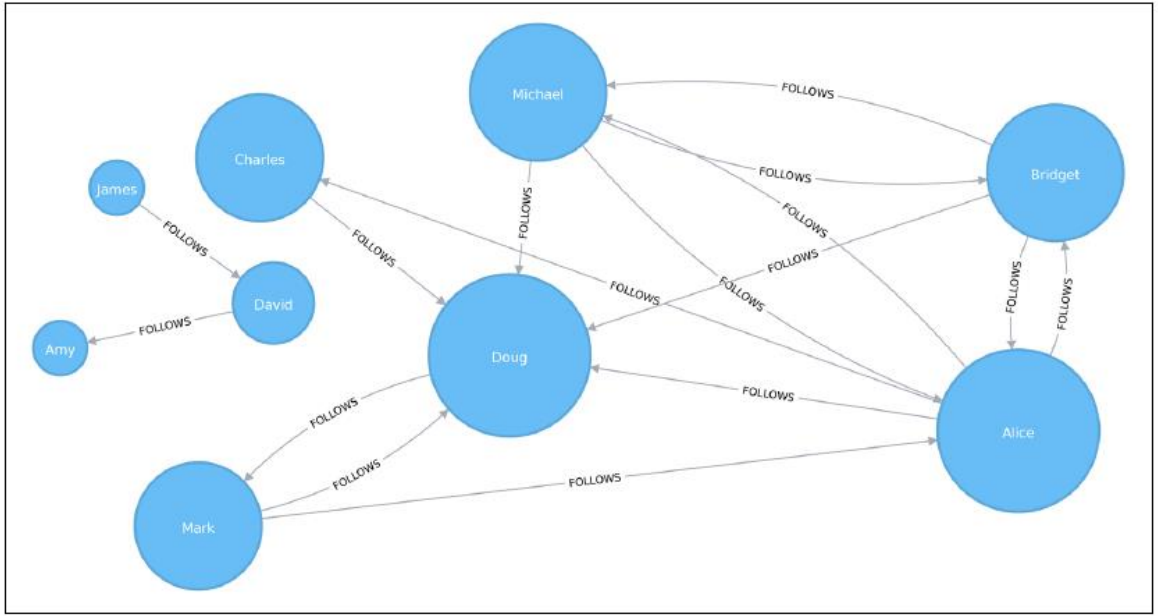
Centrality Algorithms

Closeness Centrality Variations – Wasserman Faust

As Figure shows (next slide), the results are now more representative of the closeness of nodes to the entire graph. The scores for the members of the smaller subgraph (David, Amy, and James) have been dampened, and they now have the lowest scores of all users. This makes sense as they are the most isolated nodes. This formula is more useful for detecting the importance of a node across the entire graph rather than within its own subgraph.

Centrality Algorithms

Closeness Centrality Variations – Wasserman Faust



Centrality Algorithms

Closeness Centrality Variations – Harmonic Centrality

Harmonic Centrality (also known as Valued Centrality) is a variant of Closeness Centrality, invented to solve the original problem with unconnected graphs. In “Harmony in a Small World”, M. Marchiori and V. Latora proposed this concept as a practical representation of an average shortest path.

When calculating the closeness score for each node, rather than summing the distances of a node to all other nodes, it sums the inverse of those distances. This means that infinite values become irrelevant.

Centrality Algorithms

Closeness Centrality Variations – Harmonic Centrality

The raw harmonic centrality for a node is calculated using the following formula:

$$H(u) = \sum_{v=1}^{n-1} \frac{1}{d(u,v)}$$

where u is a node; n is the number of nodes in the graph; $d(u,v)$ is the shortest-path distance between another node v and u. As with closeness centrality, we can also calculate a normalized harmonic centrality with the following formula:

$$H_{norm}(u) = \frac{\sum_{v=1}^{n-1} \frac{1}{d(u,v)}}{n-1}$$

Centrality Algorithms

Closeness Centrality Variations – Harmonic Centrality

Using the Neo4j implementation, the results from the Harmonic Centrality algorithm differ from those of the original Closeness Centrality algorithm but are similar to those from the Wasserman and Faust improvement. Either algorithm can be used when working with graphs with more than one connected component.

Centrality Algorithms

Betweenness Centrality

Sometimes the most important cog in the system is not the one with the most overt power or the highest status. Sometimes it's the middlemen that connect groups or the brokers who the most control over resources or the flow of information. Betweenness Centrality is a way of detecting the amount of influence a node has over the flow of information or resources in a graph. It is typically used to find nodes that serve as a bridge from one part of a graph to another.

Centrality Algorithms

Betweenness Centrality

The Betweenness Centrality algorithm first calculates the shortest (weighted) path between every pair of nodes in a connected graph. Each node receives a score, based on the number of these shortest paths that pass through the node. The more shortest paths that a node lies on, the higher its score. Betweenness Centrality was considered one of the “three distinct intuitive conceptions of centrality” when it was introduced by Linton C. Freeman in his 1971 paper, “A Set of Measures of Centrality Based on Betweenness”.

Graph Algorithms Mark Needham and Amy E. Hodler, O’Reilly 2019

Centrality Algorithms

Betweenness Centrality – Bridges and Control Points

A bridge in a network can be a node or a relationship. In a very simple graph, you can find them by looking for the node or relationship that, if removed, would cause a section of the graph to become disconnected. However, as that's not practical in a typical graph, we use a Betweenness Centrality algorithm. We can also measure the betweenness of a cluster by treating the group as a node. A node is considered *pivotal* for two other nodes if it lies on *every* shortest path between those nodes, as shown in Figure (next slide).

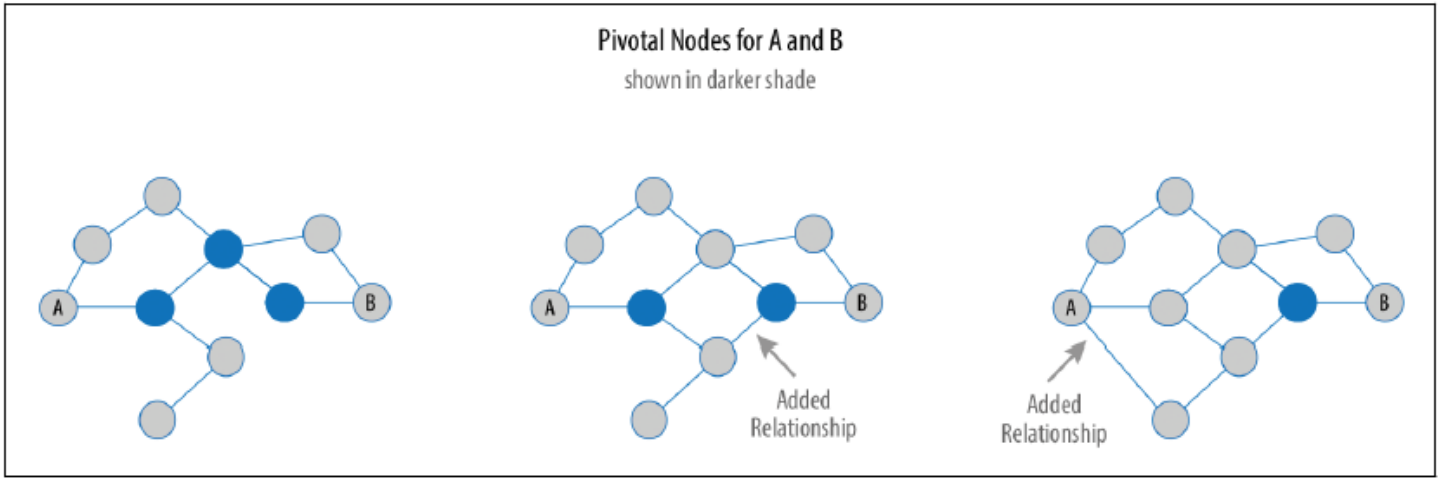
Centrality Algorithms

Betweenness Centrality – Bridges and Control Points

Pivotal nodes lie on every shortest path between two nodes. Creating more shortest paths can reduce the number of pivotal nodes for uses such as risk mitigation. Pivotal nodes play an important role in connecting other nodes—if you remove a pivotal node, the new shortest path for the original node pairs will be longer or more costly. This can be a consideration for evaluating single points of vulnerability.

Centrality Algorithms

Betweenness Centrality – Bridges and Control Points



Centrality Algorithms

Calculating Betweenness Centrality

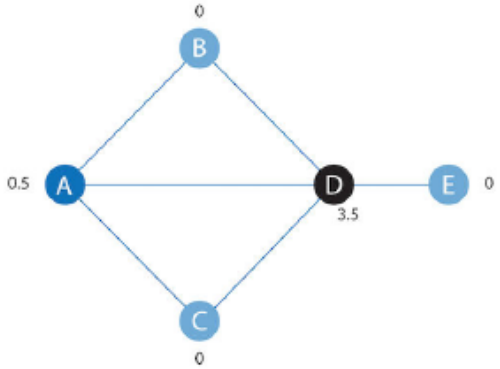
The betweenness centrality of a node is calculated by adding the results of the following formula for all shortest paths:

$$B(u) = \sum_{s \neq u \neq t} \frac{p(u)}{p}$$

where u is a node, p is the total number of shortest paths between nodes s and t, $p(u)$ is the number of shortest paths between nodes s and t that pass through node u.

Centrality Algorithms

Calculating Betweenness Centrality



Node D Calculation

Pairs with Shortest Paths Through D	Total Possible Shortest Paths for That Pair	% of Total Through D (1/Total)
A, E	1	1
B, E	1	1
C, E	1	1
B, C	2 (through D & A)	0.5
Betweenness Score		3.5

Centrality Algorithms

Calculating Betweenness Centrality - Procedure

1. For each node, find the shortest paths that go through it. a. B, C, E have no shortest paths and are assigned a value of 0.

2. For each shortest path in step 1, calculate its percentage of the total possible shortest paths for that pair.

3. Add together all the values in step 2 to find a node's betweenness centrality score. The table in Figure (previous slide) illustrates steps 2 and 3 for node D.

4. Repeat the process for each node.

Centrality Algorithms

When to Use Betweenness Centrality

Betweenness Centrality applies to a wide range of problems in real-world networks. We use it to find bottlenecks, control points, and vulnerabilities.

Note:-
Betweenness Centrality makes the assumption that all communication between nodes happens along the shortest path and with the same frequency, which isn't always the case in real life. Therefore, it doesn't give us a perfect view of the most influential nodes in a graph, but rather a good representation. Mark Newman explains this in more detail on p. 186 of *Networks: An Introduction* (Oxford University Press).

Centrality Algorithms

Betweenness Centrality – Example Use Cases

- Identifying influencers in various organizations. Powerful individuals are not necessarily in management positions but can be found in “brokerage positions” using Betweenness Centrality. Removal of such influencers can seriously destabilize the organization. This might be considered a welcome disruption by law enforcement if the organization is criminal, or could be a disaster if a business loses key staff it underestimated. More details are found in “Brokerage Qualifications in Ringing Operations”, by C. Morselli and J. Roy.

Centrality Algorithms

Betweenness Centrality – Example Use Cases

- Uncovering key transfer points in networks such as electrical grids. Counterintuitively, removal of specific bridges can actually *improve* overall robustness by "islanding" disturbances. Research details are included in "Robustness of the European Power Grids Under Intentional Attack", by R. Sole, et al.

Centrality Algorithms

Betweenness Centrality – Example Use Cases

- Helping microbloggers spread their reach on Twitter, with a recommendation engine for targeting influencers. This approach is described in a paper by S. Wu et al., “Making Recommendations in a Microblog to Improve the Impact of a Focal User”.

Centrality Algorithms

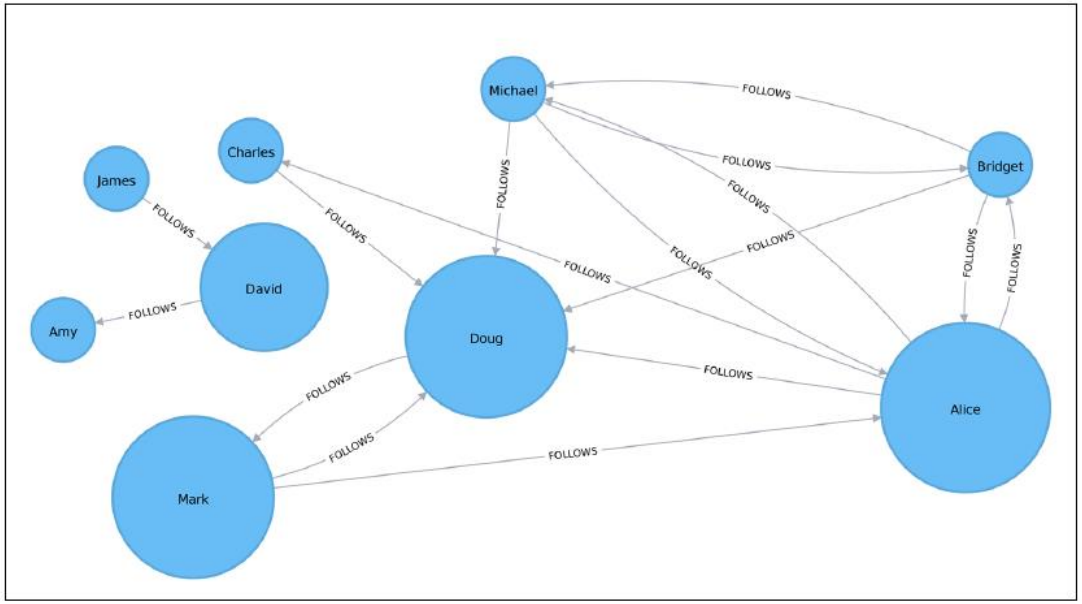
Betweenness Centrality – Neo4j

As we can see in Figure (next slide), Alice is the main broker in this network, but Mark and Doug aren't far behind. In the smaller subgraph all shortest paths go through David, so he is important for information flow among those nodes.

For large graphs, exact centrality computation isn't practical. The fastest known algorithm for exactly computing betweenness of all the nodes has a runtime proportional to the product of the number of nodes and the number of relationships. We may want to filter down to a subgraph first that works with a subset of nodes.

Centrality Algorithms

Betweenness Centrality – Neo4j



Centrality Algorithms

Betweenness Centrality Variation – RA Brandes

Recall that calculating the exact betweenness centrality on large graphs can be very expensive. We could therefore choose to use an approximation algorithm that runs much faster but still provides useful (albeit imprecise) information. The Randomized-Approximate Brandes (RA-Brandes for short) algorithm is the best-known algorithm for calculating an approximate score for betweenness centrality. Rather than calculating the shortest path between every pair of nodes, the RABrandes algorithm considers only a subset of nodes.

Due to the random nature of this algorithm, we may see different results each time that we run it. On larger graphs this randomness will have less of an impact than it does on a small sample graph.

Centrality Algorithms

Betweenness Centrality Variation – RA Brandes

Two common strategies for selecting the subset of nodes are:

Random

Nodes are selected uniformly, at random, with a defined probability of selection. The default probability is: $\frac{log_{10}(N)}{e^2}$. If the probability is 1, the algorithm works the same way as the normal Betweenness Centrality algorithm, where all nodes are loaded.

Degree

Nodes are selected randomly, but those whose degree is lower than the mean are automatically excluded (i.e., only nodes with a lot of relationships have a chance of being visited). As a further optimization, you could limit the depth used by the Shortest Path algorithm, which will then provide a subset of all the shortest paths.

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Centrality Algorithms

PageRank

PageRank is the best known of the centrality algorithms. It measures the transitive (or directional) influence of nodes. All the other centrality algorithms we discuss measure the direct influence of a node, whereas PageRank considers the influence of a node's neighbours, and their neighbours.

For example, having a few very powerful friends can make you more influential than having a lot of less powerful friends. PageRank is computed either by iteratively distributing one node's rank over its neighbours or by randomly traversing the graph and counting the frequency with which each node is hit during these walks.

Centrality Algorithms

PageRank

PageRank is named after Google cofounder Larry Page, who created it to rank websites in Google's search results.

The basic assumption is that a page with more incoming and more influential incoming links is more likely a credible source. PageRank measures the number and quality of incoming relationships to a node to determine an estimation of how important that node is. Nodes with more sway over a network are presumed to have more incoming relationships from other influential nodes.

Centrality Algorithms

PageRank - Influence

The intuition behind influence is that relationships to more important nodes contribute more to the influence of the node in question than equivalent connections to less important nodes. Measuring influence usually involves scoring nodes, often with weighted relationships, and then updating the scores over many iterations. Sometimes all nodes are scored, and sometimes a random selection is used as a representative distribution.

Note:

Keep in mind that centrality measures represent the importance of a node in comparison to other nodes. Centrality is a ranking of the potential impact of nodes, not a measure of actual impact. For example, you might identify the two people with the highest centrality in a network, but perhaps policies or cultural norms are in play that actually shift influence to others. Quantifying actual impact is an active research area to develop additional influence metrics.

Graph Algorithms Mark Needham and Amy E. Hodler, O'Reilly 2019

Centrality Algorithms - PageRank Formula

PageRank is defined in the original Google paper as follows:

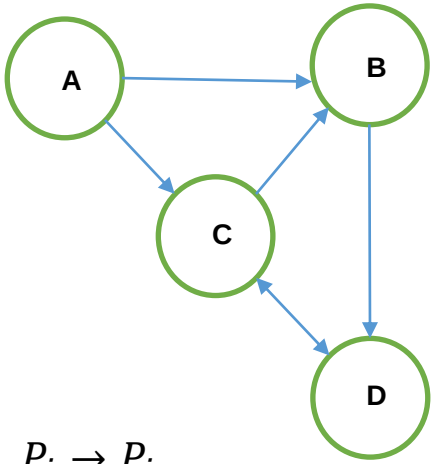
$$PR(u) = (1 - d) + d\left(\frac{PR(T1)}{C(T1)} + \cdots + \frac{PR(Tn)}{C(Tn)}\right)$$

where:

- We assume that a page u has citations from pages $T1$ to Tn
- d is a damping factor which is set between 0 and 1. It is usually set to 0.85. You
- can think of this as the probability that a user will continue clicking. This helps
- minimize rank sink, explained in the next section.
- $1 - d$ is the probability that a node is reached directly without following any relationships.
- $C(Tn)$ is defined as the out-degree of a node T.

Centrality Algorithms

PageRank Formula – Worked Example – Iteration 1



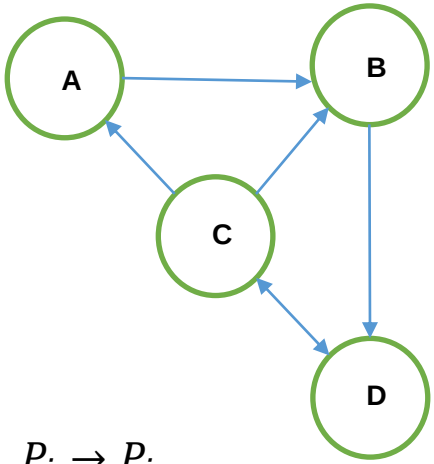
$P_i \rightarrow P_j$

	Iteration 0	Iteration 1	Iteration 2	PageRank
A	1/4			
B	1/4			
C	1/4			
D	1/4			

$$PR_{t+1}(P_i) = \frac{\sum_j PR_t(P_j)}{C(P_j)}$$

Centrality Algorithms

PageRank Formula – Worked Example – Iteration 1



	Iteration 0	Iteration 1	Iteration 2	PageRank
A	1/4	1/12		
B	1/4			
C	1/4			
D	1/4			

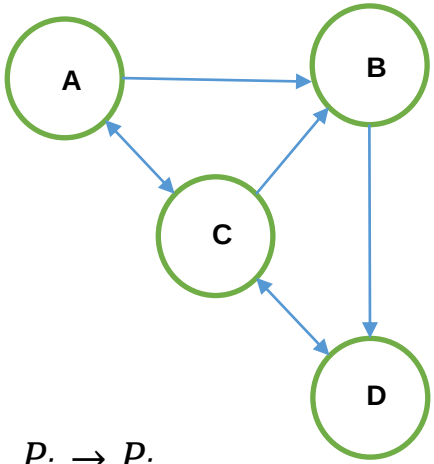
$P_i \rightarrow P_j$

$$PR_{t+1}(P_i) = \frac{\sum_j PR_t(P_j)}{C(P_j)}$$

$$\boldsymbol{PR(A) = \frac{1/4}{3} = \frac{1}{12}}$$

Centrality Algorithms

PageRank Formula – Worked Example – Iteration 1



	Iteration 0	Iteration 1	Iteration 2	PageRank
A	1/4	1/12		
B	1/4	2.5/12		
C	1/4			
D	1/4			

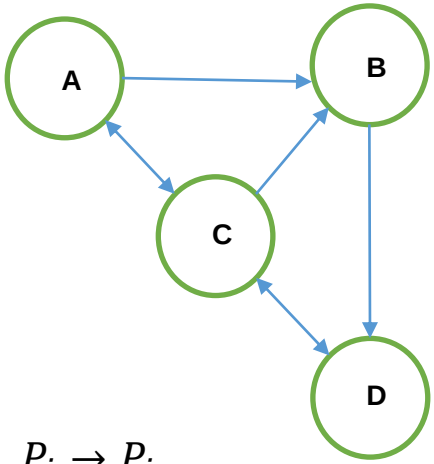
$P_i \rightarrow P_j$

$$PR_{t+1}(P_i) = \frac{\sum_j PR_t(P_j)}{C(P_j)}$$

$$\boldsymbol{PR(B) = \frac{1/4}{2} + \frac{1/4}{3} = \frac{2.5}{12}}$$

Centrality Algorithms

PageRank Formula – Worked Example – Iteration 1



	Iteration 0	Iteration 1	Iteration 2	PageRank
A	1/4	1/12		
B	1/4	2.5/12		
C	1/4	4.5/12		
D	1/4			

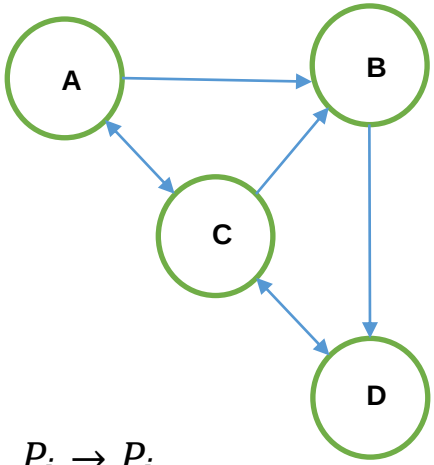
$P_i \rightarrow P_j$

$$PR_{t+1}(P_i) = \frac{\sum_j PR_t(P_j)}{C(P_j)}$$

$$\boldsymbol{PR(C) = \frac{1/4}{2} + \frac{1/4}{1} = \frac{4.5}{12}}$$

Centrality Algorithms

PageRank Formula – Worked Example – Iteration 1



	Iteration 0	Iteration 1	Iteration 2	PageRank
A	1/4	1/12		
B	1/4	2.5/12		
C	1/4	4.5/12		
D	1/4	4/12		

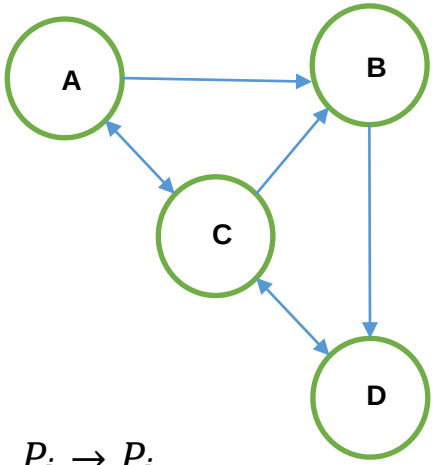
$P_i \rightarrow P_j$

$$PR_{t+1}(P_i) = \frac{\sum_j PR_t(P_j)}{C(P_j)}$$

$$\boldsymbol{PR(D) = \frac{1/4}{3} + \frac{1/4}{1} = \frac{4}{12}}$$

Centrality Algorithms

PageRank Formula – Worked Example – Iteration 2



	Iteration 0	Iteration 1	Iteration 2	PageRank
A	1/4	1/12	1.5/12	
B	1/4	2.5/12		
C	1/4	4.5/12		
D	1/4	4/12		

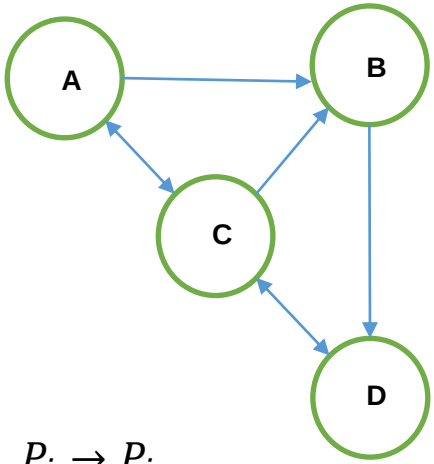
$P_i \rightarrow P_j$

$$PR_{t+1}(P_i) = \frac{\sum_j PR_t(P_j)}{C(P_j)}$$

$$\boldsymbol{PR(A) = \frac{4.5/12}{3} = \frac{1.5}{12}}$$

Centrality Algorithms

PageRank Formula – Worked Example – Iteration 2



	Iteration 0	Iteration 1	Iteration 2	PageRank
A	1/4	1/12	1.5/12	
B	1/4	2.5/12	2/12	
C	1/4	4.5/12		
D	1/4	4/12		

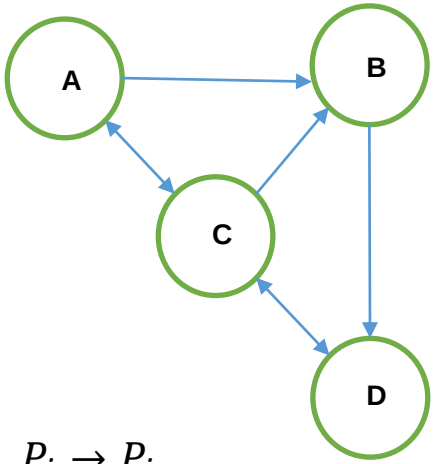
$P_i \rightarrow P_j$

$$PR_{t+1}(P_i) = \frac{\sum_j PR_t(P_j)}{C(P_j)}$$

$$\boldsymbol{PR(B) = \frac{1/12}{2} + \frac{4.5/12}{3} = \frac{2}{12}}$$

Centrality Algorithms

PageRank Formula – Worked Example – Iteration 2



	Iteration 0	Iteration 1	Iteration 2	PageRank
A	1/4	1/12	1.5/12	
B	1/4	2.5/12	2/12	
C	1/4	4.5/12	4.5/12	
D	1/4	4/12		

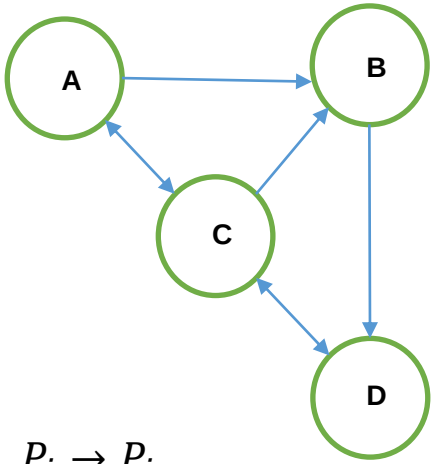
$P_i \rightarrow P_j$

$$PR_{t+1}(P_i) = \frac{\sum_j PR_t(P_j)}{C(P_j)}$$

$$\boldsymbol{PR(C) = \frac{1/12}{2} + \frac{4/12}{1} = \frac{4.5}{12}}$$

Centrality Algorithms

PageRank Formula – Worked Example – Iteration 2



	Iteration 0	Iteration 1	Iteration 2	PageRank
A	1/4	1/12	1.5/12	
B	1/4	2.5/12	2/12	
C	1/4	4.5/12	4.5/12	
D	1/4	4/12	4/12	

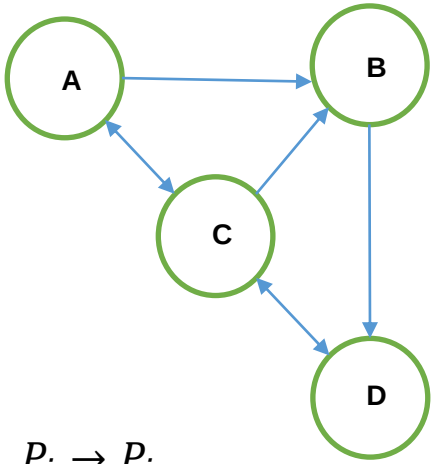
$P_i \rightarrow P_j$

$$PR_{t+1}(P_i) = \frac{\sum_j PR_t(P_j)}{C(P_j)}$$

$$\boldsymbol{PR(D) = \frac{4.5/12}{3} + \frac{2.5/12}{1} = \frac{4}{12}}$$

Centrality Algorithms

PageRank Formula – Worked Example – PageRank



$P_i \rightarrow P_j$

	Iteration 0	Iteration 1	Iteration 2	PageRank
A	1/4	1/12	1.5/12	1
B	1/4	2.5/12	2/12	2
C	1/4	4.5/12	4.5/12	4
D	1/4	4/12	4/12	3

$$PR_{t+1}(P_i) = \frac{\sum_j PR_t(P_j)}{C(P_j)}$$

Centrality Algorithms

Iteration, Random Surfers, and Rank Sinks

PageRank is an iterative algorithm that runs either until scores converge or until a set number of iterations is reached. Conceptually, PageRank assumes there is a web surfer visiting pages by following links or by using a random URL. A damping factor _d _ defines the probability that the next click will be through a link. You can think of it as the probability that a surfer will become bored and randomly switch to another page. A PageRank score represents the likelihood that a page is visited through an incoming link and not randomly..

Centrality Algorithms

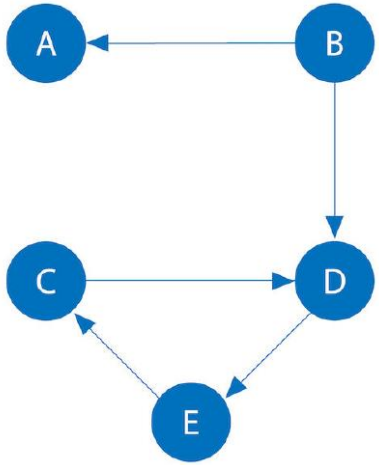
Iteration, Random Surfers, and Rank Sinks

A node, or group of nodes, without outgoing relationships (also called a *dangling node*) can monopolize the PageRank score by refusing to share. This is known as a *rank sink*. You can imagine this as a surfer that gets stuck on a page, or a subset of pages, with no way out. Another difficulty is created by nodes that point only to each other in a group. Circular references cause an increase in their ranks as the surfer bounces back and forth among the nodes.

Centrality Algorithms

Iteration, Random Surfers, and Rank Sinks

Rank Sinks Monopolize Rank Scores



A is a dangling node with no outgoing relationships. Teleportation is used to overcome dead ends.

C, D, and E are circular references with no way out of the group. A dampening factor is used to introduce random node visits.

Centrality Algorithms

Iteration, Random Surfers, and Rank Sinks

There are two strategies used to avoid rank sinks. First, when a node is reached that has no outgoing relationships, PageRank assumes outgoing relationships to all nodes. traversing these invisible links is sometimes called *teleportation*. Second, the damping factor provides another opportunity to avoid sinks by introducing a probability for direct link versus random node visitation. When you set *d* to 0.85, a completely random node is visited 15% of the time.

Centrality Algorithms

Iteration, Random Surfers, and Rank Sinks

Although the original formula recommends a damping factor of 0.85, its initial use was on the World Wide Web with a power-law distribution of links (most pages have very few links and a few pages have many). Lowering the damping factor decreases the likelihood of following long relationship paths before taking a random jump. In turn, this increases the contribution of a node's immediate predecessors to its score and rank. If you see unexpected results from PageRank, it is worth doing some exploratory analysis of the graph to see if any of these problems are the cause.

Read Ian Rogers's article, "The Google PageRank Algorithm and How It Works" to learn more http://ianrogers.uk/google-page-rank/

Centrality Algorithms

When to Use PageRank

PageRank is now used in many domains outside web indexing. Use this algorithm whenever you're looking for broad influence over a network. For instance, if you're looking to target a gene that has the highest overall impact to a biological function, it may not be the most connected one. It may, in fact, be the gene with the most relationships with other, more significant functions.

Centrality Algorithms

PageRank - Example Use Cases

- Presenting users with recommendations of other accounts that they may wish to follow (Twitter uses Personalized PageRank for this). The algorithm is run over a graph that contains shared interests and common connections. The approach is described in more detail in the paper "WTF: The Who to Follow Service at Twitter", by P. Gupta et al.

- Predicting traffic flow and human movement in public spaces or streets. The algorithm is run over a graph of road intersections, where the PageRank score reflects the tendency of people to park, or end their journey, on each street. This is described in more detail in "Self-Organized Natural Roads for Predicting Traffic Flow: A Sensitivity Study", a paper by B. Jiang, S. Zhao, and J. Yin.

Centrality Algorithms

PageRank - Example Use Cases

- As part of anomaly and fraud detection systems in the healthcare and insurance industries. PageRank helps reveal doctors or providers that are behaving in an unusual manner, and the scores are then fed into a machine learning algorithm.

David Gleich describes many more uses for the algorithm in his paper, "PageRank Beyond the Web".

Centrality Algorithms

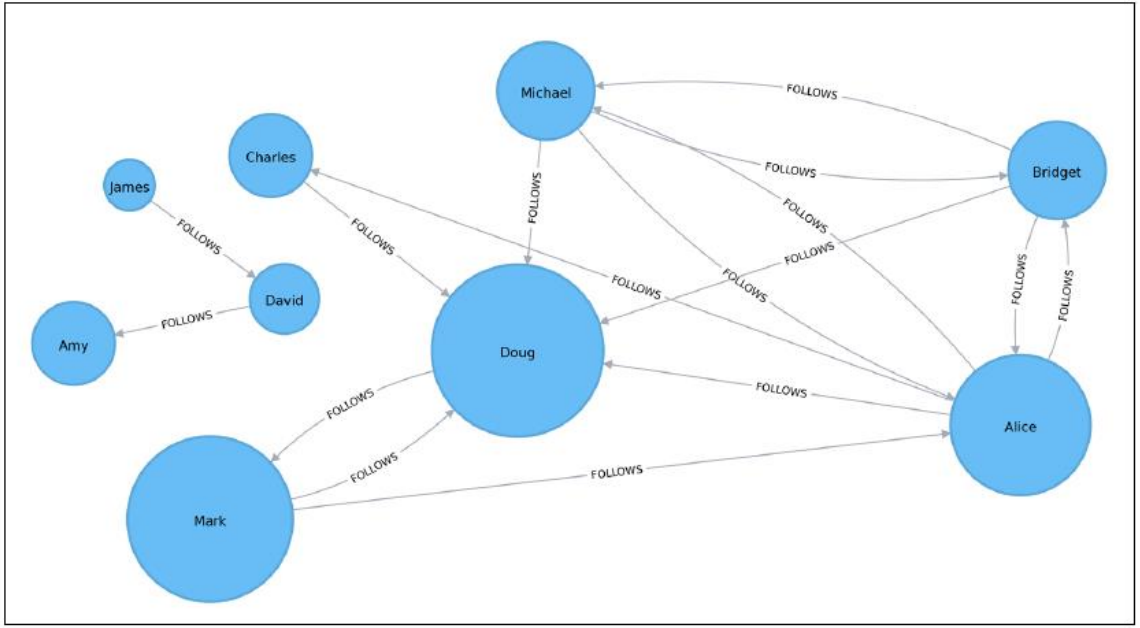
PageRank - Neo4j

Note: PageRank implementations vary, so they can produce different scoring even when the ordering is the same. Neo4j initializes nodes using a value of 1 minus the dampening factor whereas Spark uses a value of 1. In this case, the relative rankings (the goal of PageRank) are identical but the underlying score values used to reach those results are different.

From Figure (next slide). As we might expect, Doug has the highest PageRank because he is followed by all other users in his subgraph. Although Mark only has one follower, that follower is Doug, so Mark is also considered important in this graph. It's not only the number of followers that is important, but also the importance of those followers.

Centrality Algorithms

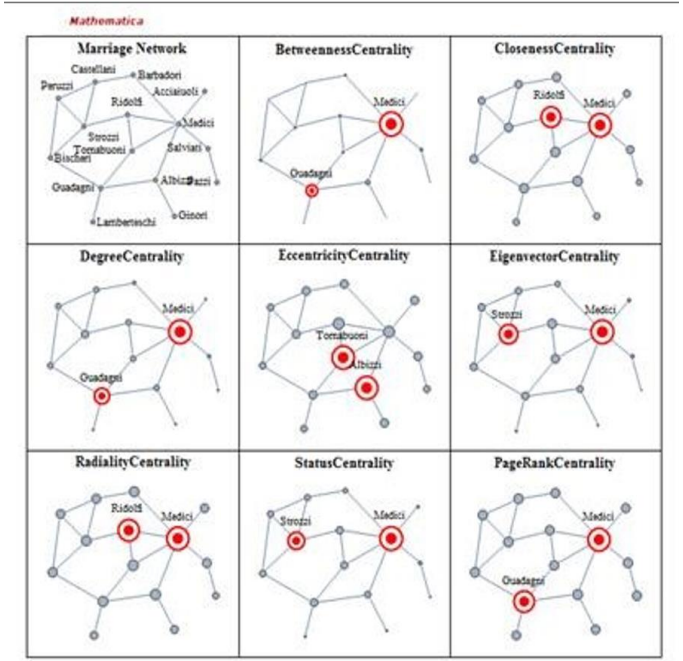
PageRank - Neo4j



Centrality Algorithms

In Practice 1

The **Medici family**, also known as the House of **Medici**, was the Italian **family** that ruled **Florence**, and later Tuscany, during most of the period from 1434 to 1737, except for two brief intervals (from 1494 to 1512, and from 1527 to 1530). The Medici family did not have greatest wealth or most seats in the legislature, yet it rose to power. Through **marriages**, the Medici family had a position of centrality in the social network, crucial for communication, brokering deals, etc.



Centrality Algorithms

Summary

There are many wide-ranging uses for centrality algorithms and exploration is encouraged for a variety of analyses. You can apply centrality algorithms to locate optimal touch points for disseminating information, find the hidden brokers that control the flow of resources and uncover the indirect power players lurking in the shadows.