CHAPTER 5

Centrality Algorithms

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.

We'll cover the following algorithms:

- 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

Different centrality algorithms can produce significantly different results based on what they were created to measure. When you see suboptimal answers, it's best to check the algorithm you've used is aligned to its intended purpose.

We'll explain how these algorithms work and show examples in Spark and Neo4j. Where an algorithm is unavailable on one platform or where the differences are unimportant, we'll provide just one platform example.

Figure 5-1 shows the differences between the types of questions centrality algorithms can answer, and Table 5-1 is a quick reference for what each algorithm calculates with an example use.

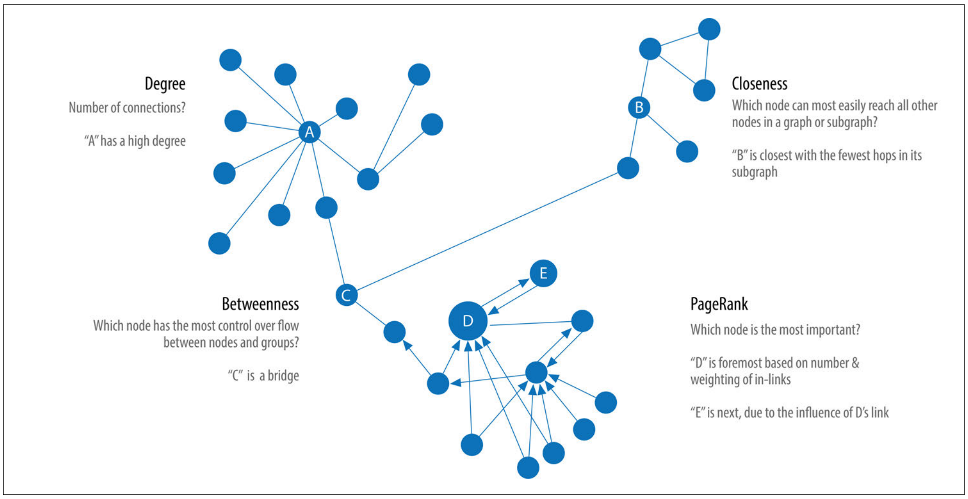

Degree
Number of connections?
"A" has a high degree
A
Closeness
Which node can most easily reach all other nodes in a graph or subgraph?
"B" is closest with the fewest hops in its subgraph
B
Betweenness
Which node has the most control over flow between nodes and groups?
"C" is a bridge
C
PageRank
Which node is the most important?
"D" is foremost based on number & weighting of in-links
"E" is next, due to the influence of D's link
D
E

Figure 5-1. Representative centrality algorithms and the types of questions they answer

Table 5-1. Overview of centrality algorithms

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

Several of the centrality algorithms calculate shortest paths between every pair of nodes. This works well for small- to medium-sized graphs but for large graphs can be computationally prohibitive. To avoid long runtimes on larger graphs, some algorithms (for example, Betweenness Centrality) have approximating versions.

First, we'll describe the dataset for our examples and walk through importing the data into Apache Spark and Neo4j. Each algorithm is covered in the order listed in Table 5-1. We'll start with a short description of the algorithm and, when warranted, information on how it operates. Variations of algorithms already covered will include less detail. Most sections also include guidance on when to use the related algorithm. We demonstrate example code using a sample dataset at the end of each section.

Let's get started!

Example Graph Data: The 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 chapter are run against a small Twitter-like graph. You can download the nodes and relationships files we'll use to create our graph from the book's GitHub repository.

Table 5-2. social-nodes.csv

id
Alice
Bridget
Charles
Doug
Mark
Michael
David
Amy
James

Table 5-3. social-relationships.csv

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

Figure 5-2 illustrates the graph that we want to construct.

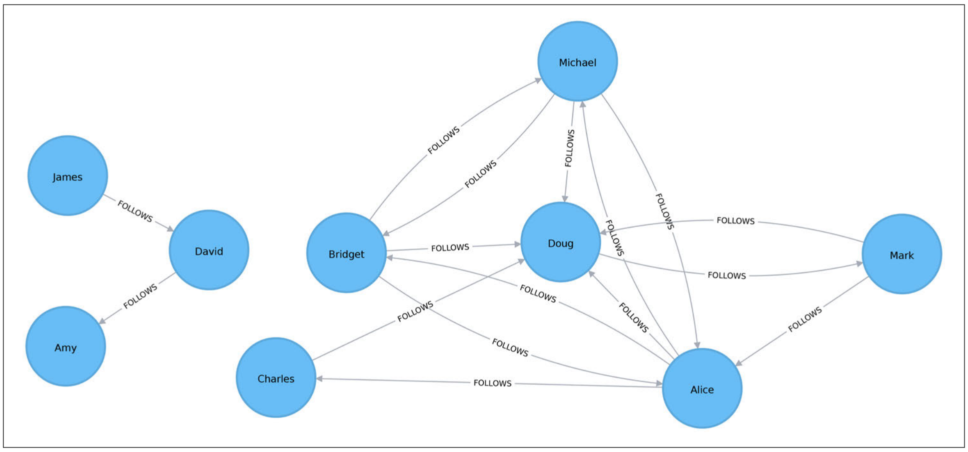


Figure 5-2. The graph model

We have one larger set of users with connections between them and a smaller set with no connections to that larger group.

Let's create graphs in Spark and Neo4j based on the contents of those CSV files.

Importing the Data into Apache Spark

First, we'll import the required packages from Spark and the GraphFrames package:

```
from graphframes import *
from pyspark import SparkContext
```

We can write the following code to create a GraphFrame based on the contents of the CSV files:

```
v = spark.read.csv("data/social-nodes.csv", header=True)
e = spark.read.csv("data/social-relationships.csv", header=True)
g = GraphFrame(v, e)
```

Importing the Data into Neo4j

Next, we'll load the data for Neo4j. The following query imports nodes:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "social-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (:User {id: row.id})
```

And this query imports relationships:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "social-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (source:User {id: row.src})
MATCH (destination:User {id: row.dst})
MERGE (source)-[:FOLLOWS]->(destination)
```

Now that our graphs are loaded, it's on to the algorithms!

Degree Centrality

Degree Centrality is the simplest of the algorithms that we'll cover in this book. 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 cold circulating in their 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.

Figure 5-3 illustrates the difference looking at the actual distribution of connections among subreddit topics. If you simply took the average, you'd assume most topics have 10 connections, whereas in fact most topics only have 2 connections.

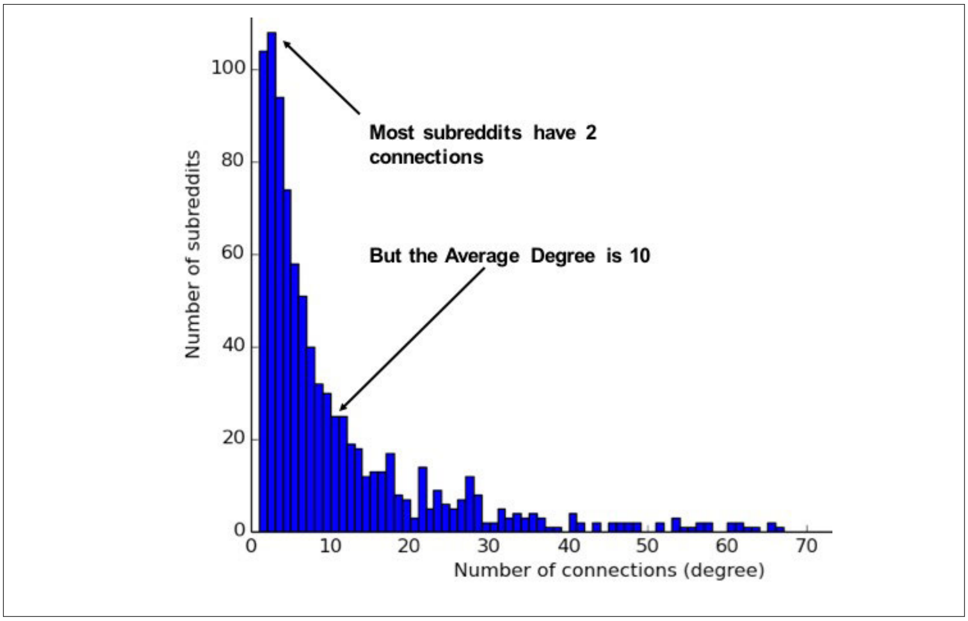


Figure 5-3. This mapping of subreddit degree distribution by Jacob Silterrapa provides an example of how the average does not often reflect the actual distribution in networks. CC BY-SA 3.0.

These measures are used to categorize network types such as the scale-free or small-world networks that were discussed in Chapter 2. They also provide a quick measure to help estimate the potential for things to spread or ripple throughout a network.

When Should I Use Degree Centrality?

Use Degree Centrality if you're attempting to analyze 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.

Example use cases include:

- 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.
- 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".

Degree Centrality with Apache Spark

Now we'll execute the Degree Centrality algorithm with the following code:

```
total_degree = g.degrees
in_degree = g.inDegrees
out_degree = g.outDegrees

(total_degree.join(in_degree, "id", how="left")
 .join(out_degree, "id", how="left")
 .fillna(0)
 .sort("inDegree", ascending=False)
 .show())
```

We first calculate the total, in, and out degrees. Then we join those DataFrames together, using a left join to retain any nodes that don't have incoming or outgoing relationships. If nodes don't have relationships we set that value to `0` using the `fillna` function.

Here's the result of running the code in pyspark:

id	degree	inDegree	outDegree
Doug	6	5	1
Alice	7	3	4
Michael	5	2	3
Bridget	5	2	3
Charles	2	1	1
Mark	3	1	2
David	2	1	1
Amy	1	1	0
James	1	0	1

We can see in Figure 5-4 that 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!

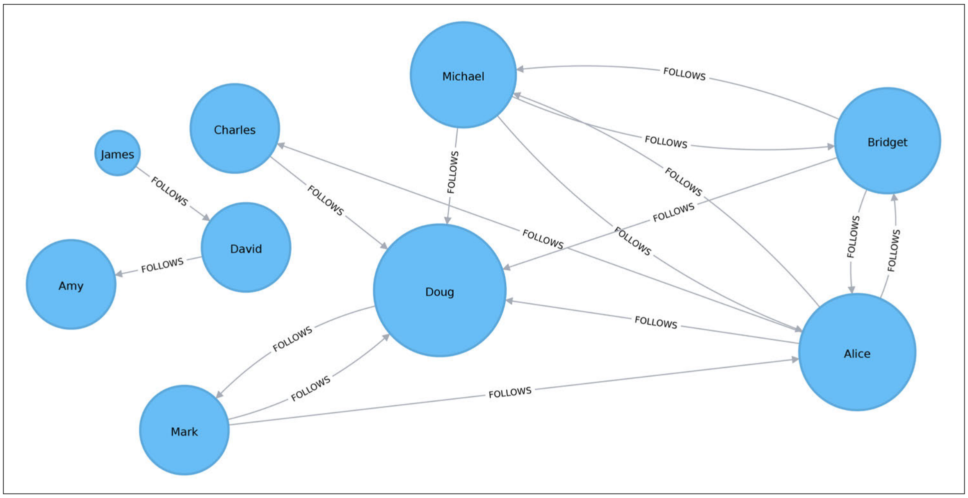


Figure 5-4. Visualization of degree centrality

If we were creating a page showing the most-followed users or wanted to suggest people to follow, we could use this algorithm to identify those people.

Some data may contain very dense nodes with lots of relationships. These don't add much additional information and can skew some results or add computational complexity. You may want to filter out these dense notes by using a subgraph, or use a projection to summarize the relationships as weights.

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.

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:

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

When Should I 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 behavioral analyses.

Example use cases include:

- Uncovering individuals in very favorable 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 keyphrase extraction process. This process is described by F. Boudin in "A Comparison of Centrality Measures for Graph-Based Keyphrase Extraction".

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. This means that we'll end up with 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 will be shown after the next example.

Closeness Centrality with Apache Spark

Apache Spark doesn't have a built-in algorithm for Closeness Centrality, but we can write our own using the `aggregateMessages` framework that we introduced in the "Shortest Path (Weighted) with Apache Spark" on page 54 in the previous chapter.

Before we create our function, we'll import some libraries that we'll use:

```
from graphframes.lib import AggregateMessages as AM
from pyspark.sql import functions as F
from pyspark.sql.types import *
from operator import itemgetter
```

We'll also create a few user-defined functions that we'll need later:

```
def collect_paths(paths):
    return F.collect_set(paths)


collect_paths_udf = F.udf(collect_paths, ArrayType(StringType()))

paths_type = ArrayType(
    StructType([StructField("id", StringType()), StructField("distance",


def flatten(ids):
    flat_list = [item for sublist in ids for item in sublist]
    return list(dict(sorted(flat_list, key=itemgetter(0))).items())


flatten_udf = F.udf(flatten, paths_type)


def new_paths(paths, id):
    paths = [{"id": col1, "distance": col2 + 1} for col1,
                            col2 in paths if col1 != id]
    paths.append({"id": id, "distance": 1})
    return paths


new_paths_udf = F.udf(new_paths, paths_type)


def merge_paths(ids, new_ids, id):
    joined_ids = ids + (new_ids if new_ids else [])
    merged_ids = [(col1, col2) for col1, col2 in joined_ids if col1 != id]
    best_ids = dict(sorted(merged_ids, key=itemgetter(1), reverse=True))
    return [{"id": col1, "distance": col2} for col1, col2 in best_ids.items()]


merge_paths_udf = F.udf(merge_paths, paths_type)
```

```
def calculate_closeness(ids):
    nodes = len(ids)
    total_distance = sum([col2 for col1, col2 in ids])
    return 0 if total_distance == 0 else nodes * 1.0 / total_distance


closeness_udf = F.udf(calculate_closeness, DoubleType())
```

And now for the main body that calculates the closeness centrality for each node:

```
vertices = g.vertices.withColumn("ids", F.array())
cached_vertices = AM.getCachedDataFrame(vertices)
g2 = GraphFrame(cached_vertices, g.edges)

for i in range(0, g2.vertices.count()):
    msg_dst = new_paths_udf(AM.src["ids"], AM.src["id"])
    msg_src = new_paths_udf(AM.dst["ids"], AM.dst["id"])
    agg = g2.aggregateMessages(F.collect_set(AM.msg).alias("agg"),
                               sendToSrc=msg_src, sendToDst=msg_dst)
    res = agg.withColumn("newIds", flatten_udf("agg")).drop("agg")
    new_vertices = (g2.vertices.join(res, on="id", how="left_outer")
                    .withColumn("mergedIds", merge_paths_udf("ids", "newIds",
                    "id")).drop("ids", "newIds")
                    .withColumnRenamed("mergedIds", "ids"))
    cached_new_vertices = AM.getCachedDataFrame(new_vertices)
    g2 = GraphFrame(cached_new_vertices, g2.edges)

(g2.vertices
 .withColumn("closeness", closeness_udf("ids"))
 .sort("closeness", ascending=False)
 .show(truncate=False))
```

If we run that we'll see the following output:

id	ids	closeness
Doug	[[Charles, 1], [Mark, 1], [Alice, 1], [Bridget, 1], [Michael, 1]]	1.0
Alice	[[Charles, 1], [Mark, 1], [Bridget, 1], [Doug, 1], [Michael, 1]]	1.0
David	[[James, 1], [Amy, 1]]	1.0
Bridget	[[Charles, 2], [Mark, 2], [Alice, 1], [Doug, 1], [Michael, 1]]	0.7142857142857143
Michael	[[Charles, 2], [Mark, 2], [Alice, 1], [Doug, 1], [Bridget, 1]]	0.7142857142857143
James	[[Amy, 2], [David, 1]]	0.6666666666666666
Amy	[[James, 2], [David, 1]]	0.6666666666666666
Mark	[[Bridget, 2], [Charles, 2], [Michael, 2], [Doug, 1], [Alice, 1]]	0.625
Charles	[[Bridget, 2], [Mark, 2], [Michael, 2], [Doug, 1], [Alice, 1]]	0.625

Alice, Doug, and David are the most closely connected nodes in the graph with a 1.0 score, which means each directly connects to all nodes in their part of the graph. Figure 5-5 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.

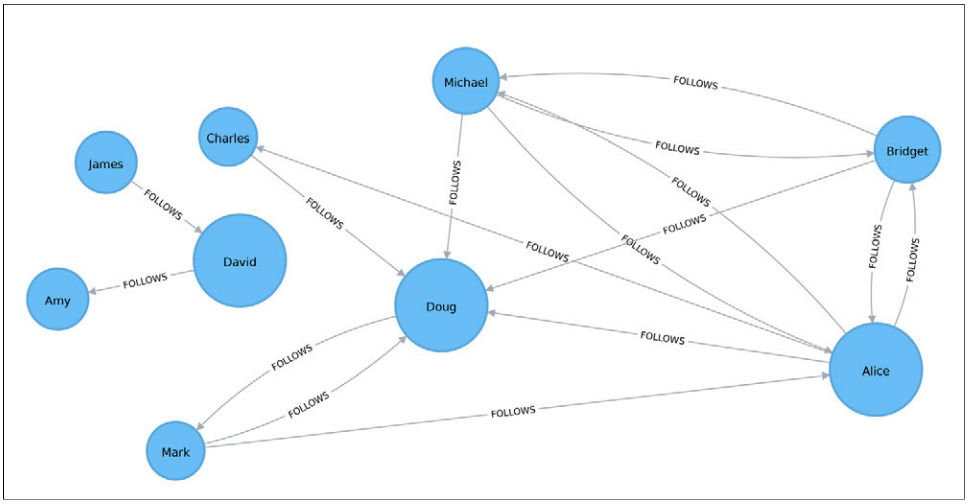


Figure 5-5. Visualization of closeness centrality

Closeness Centrality with Neo4j

Neo4j's implementation of Closeness Centrality uses the following formula:

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

where:

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

A call to the following procedure will calculate the closeness centrality for each of the nodes in our graph:

```
CALL algo.closeness.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id, centrality
ORDER BY centrality DESC
```

Running this procedure gives the following output:

user	centrality
Alice	1.0
Doug	1.0

user	centrality
David	1.0
Bridget	0.7142857142857143
Michael	0.7142857142857143
Amy	0.6666666666666666
James	0.6666666666666666
Charles	0.625
Mark	0.625

We get the same results as with the Spark algorithm, but, as before, the score represents their 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, and in the next two sections we'll learn about a few variations of the Closeness Centrality algorithm that do this.

Closeness Centrality Variation: Wasserman and 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:

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

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.

We can tell the Closeness Centrality procedure to use this formula by passing the parameter `improved: true`.

The following query executes Closeness Centrality using the Wasserman and Faust formula:

```
CALL algo.closeness.stream("User", "FOLLOWS", {improved: true})
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

The procedure gives the following result:

user	centrality
Alice	0.5
Doug	0.5
Bridget	0.35714285714285715
Michael	0.35714285714285715
Charles	0.3125
Mark	0.3125
David	0.125
Amy	0.08333333333333333
James	0.08333333333333333

As Figure 5-6 shows, 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.

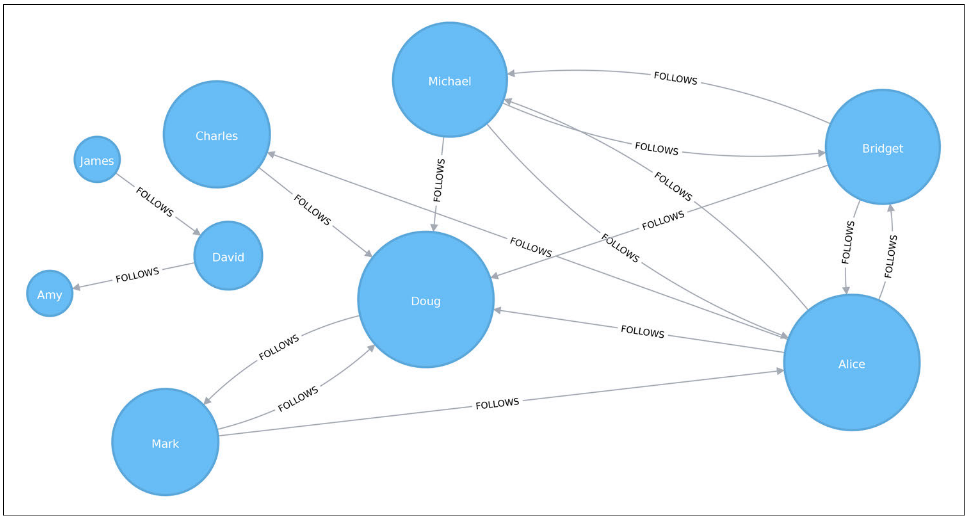


Figure 5-6. Visualization of closeness centrality

In the next section we'll learn about the Harmonic Centrality algorithm, which achieves similar results using another formula to calculate closeness.

Closeness Centrality Variation: 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.

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:

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

In this formula, ∞ values are handled cleanly.

Harmonic Centrality with Neo4j

The following query executes the Harmonic Centrality algorithm:

```
CALL algo.closeness.harmonic.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

Running this procedure gives the following result:

user	centrality
Alice	0.625
Doug	0.625
Bridget	0.5
Michael	0.5
Charles	0.4375
Mark	0.4375
David	0.25
Amy	0.1875
James	0.1875

The results from this 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.

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.

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".

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 5-7.

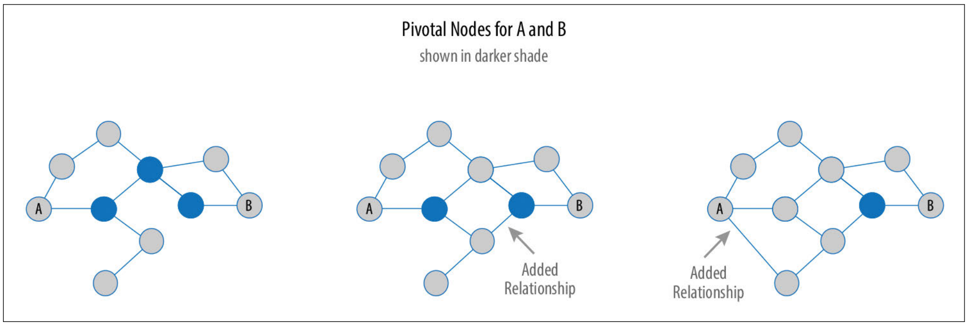


Figure 5-7. Pivotal nodes lie on every shortest path between two nodes. Creating more shortest paths can reduce the number of pivotal nodes for uses such as risk mitigation.

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.

Calculating betweenness centrality

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

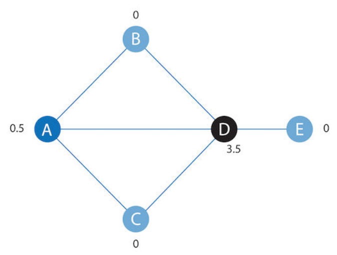
$$B(u) = \sum_{s \neq u \neq t} \frac{p(u)}{p}$$

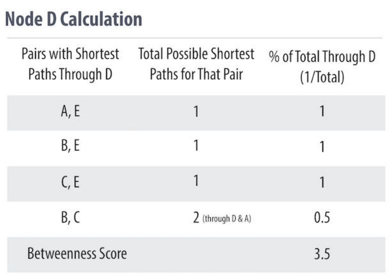
where:

- 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*.

Figure 5-8 illustrates the steps for working out 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

Figure 5-8. Basic concepts for calculating betweenness centrality

Here's the 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 5-8 illustrates steps 2 and 3 for node D.
4. Repeat the process for each node.

When Should I 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.

Example use cases include:

- 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.

- 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. Solé, et al.
- 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".

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 in *Networks: An Introduction* (Oxford University Press, p186).

Betweenness Centrality with Neo4j

Spark doesn't have a built-in algorithm for Betweenness Centrality, so we'll demonstrate this algorithm using Neo4j. A call to the following procedure will calculate the betweenness centrality for each of the nodes in our graph:

```
CALL algo.betweenness.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id  AS user, centrality
ORDER BY centrality DESC
```

Running this procedure gives the following result:

user	centrality
Alice	10.0
Doug	7.0
Mark	7.0
David	1.0
Bridget	0.0
Charles	0.0
Michael	0.0
Amy	0.0
James	0.0

As we can see in Figure 5-9, 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.

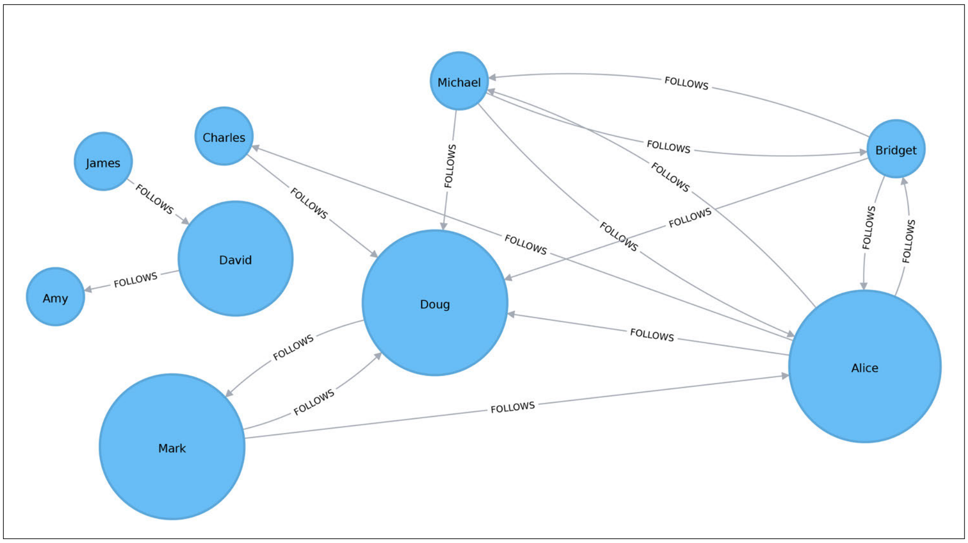


Figure 5-9. Visualization of betweenness centrality

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 or use (described in the next section) that works with a subset of nodes.

We can join our two disconnected components together by introducing a new user called Jason, who follows and is followed by people from both groups of users:

```
WITH ["James", "Michael", "Alice", "Doug", "Amy"] AS existingUsers

MATCH (existing:User) WHERE existing.id IN existingUsers
MERGE (newUser:User {id: "Jason"})

MERGE (newUser)<-[:FOLLOWS]-(existing)
MERGE (newUser)-[:FOLLOWS]->(existing)
```

If we rerun the algorithm we'll see this output:

user	centrality
Jason	44.33333333333333
Doug	18.333333333333332
Alice	16.666666666666664
Amy	8.0
James	8.0
Michael	4.0
Mark	2.1666666666666665
David	0.5
Bridget	0.0
Charles	0.0

Jason has the highest score because communication between the two sets of users will pass through him. Jason can be said to act as a *local bridge* between the two sets of users, as illustrated in Figure 5-10.

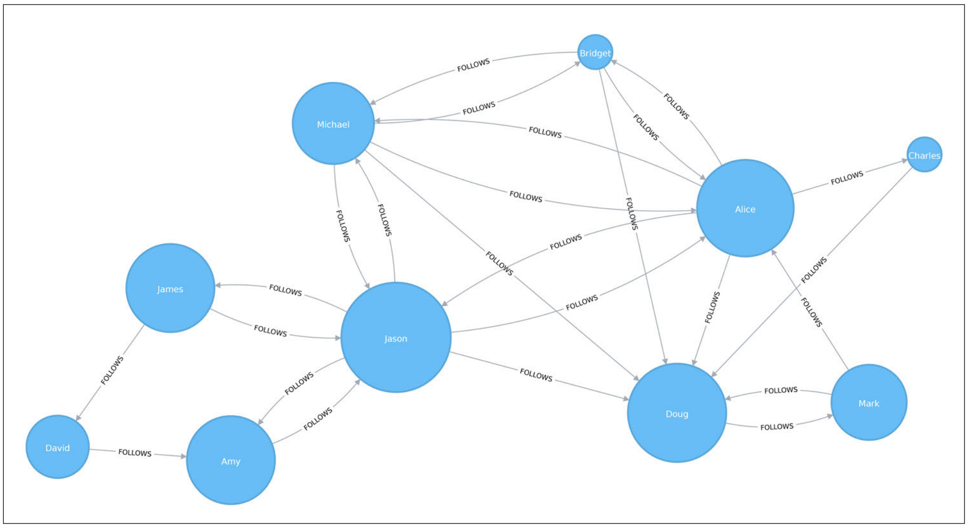


Figure 5-10. Visualization of betweenness centrality with Jason

Before we move on to the next section, let's reset our graph by deleting Jason and his relationships:

```
MATCH (user:User {id: "Jason"})
DETACH DELETE user
```

Betweenness Centrality Variation: Randomized-Approximate 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 RA-Brandes algorithm considers only a subset of nodes. 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: $\frac{log10(N)}{e^2}$. 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.

Approximation of Betweenness Centrality with Neo4j

The following query executes the RA-Brandes algorithm using the random selection method:

```
CALL algo.betweenness.sampled.stream("User", "FOLLOWS", {strategy:"degree"})
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

Running this procedure gives the following result:

user	centrality
Alice	9.0
Mark	9.0
Doug	4.5

user	centrality
David	2.25
Bridget	0.0
Charles	0.0
Michael	0.0
Amy	0.0
James	0.0

Our top influencers are similar to before, although Mark now has a higher ranking than Doug.

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 our small sample graph.

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 neighbors, and their neighbors. 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 neighbors or by randomly traversing the graph and counting the frequency with which each node is hit during these walks.

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.

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.

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.

The PageRank Formula

PageRank is defined in the original Google paper as follows:

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

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*.

Figure 5-11 walks through a small example of how PageRank will continue to update the rank of a node until it converges or meets the set number of iterations.

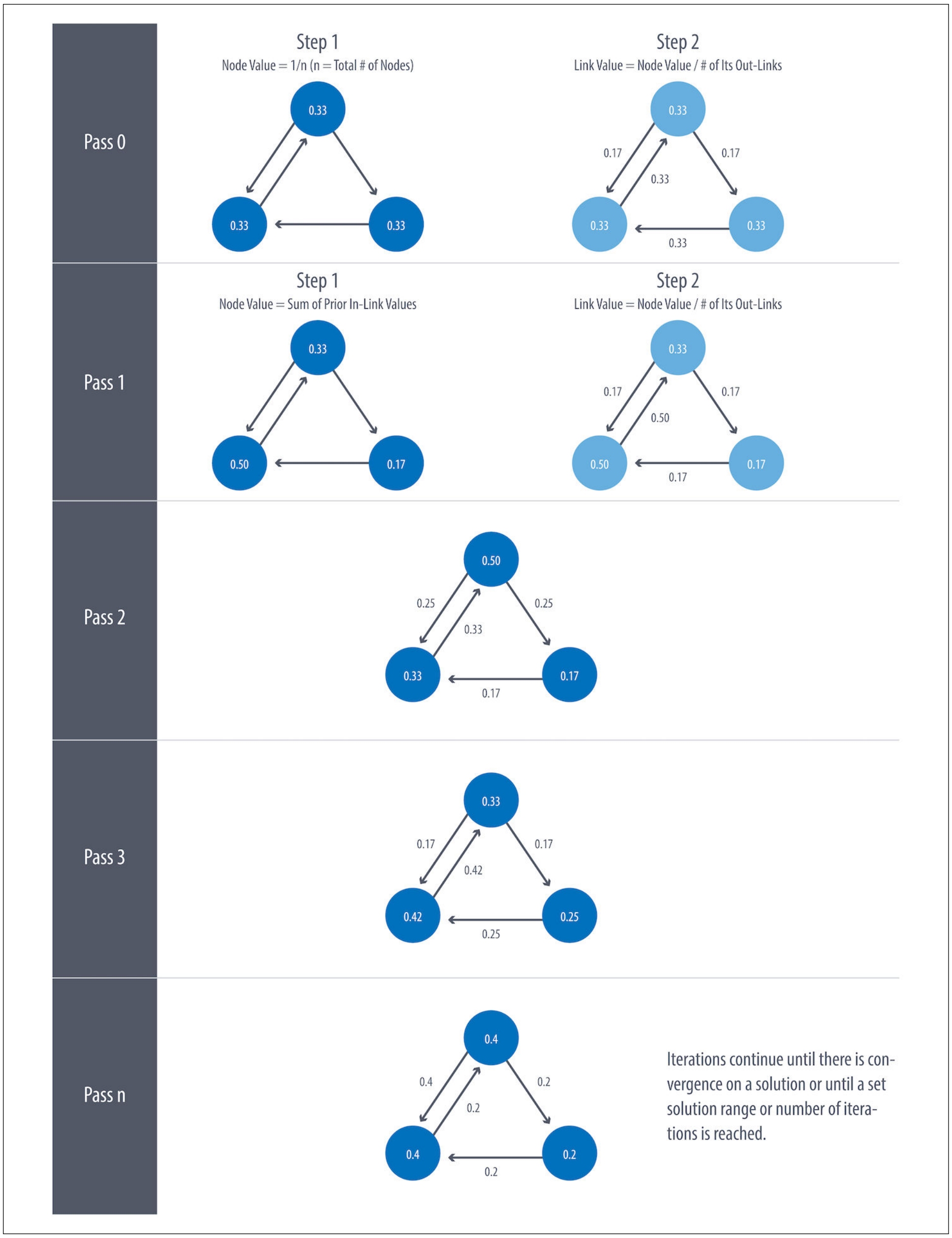

Pass 0
Step 1
Node Value = 1/n (n = Total # of Nodes)
0.33
0.33
0.33
Step 2
Link Value = Node Value / # of Its Out-Links
0.33
0.17
0.17
0.33
0.33
0.33
0.33
Pass 1
Step 1
Node Value = Sum of Prior In-Link Values
0.33
0.50
0.17
Step 2
Link Value = Node Value / # of Its Out-Links
0.33
0.17
0.17
0.50
0.50
0.17
0.17
Pass 2
0.50
0.25
0.25
0.33
0.33
0.17
0.17
Pass 3
0.33
0.17
0.17
0.42
0.42
0.25
0.25
Pass n
0.4
0.4
0.2
0.2
0.4
0.2
0.2
Iterations continue until there is convergence on a solution or until a set solution range or number of iterations is reached.

Figure 5-11. Each iteration of PageRank has two calculation steps: one to update node values and one to update link values.

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.

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. These situations are portrayed in Figure 5-12.

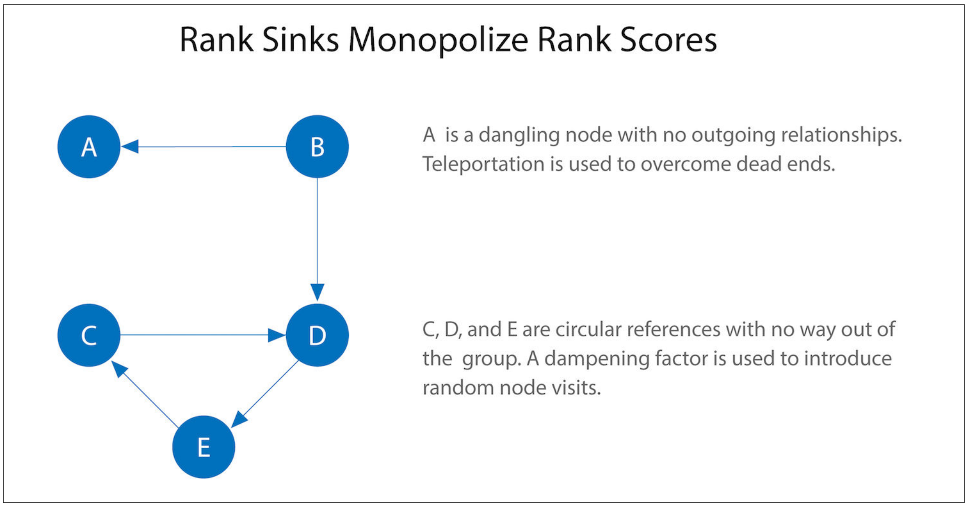

Rank Sinks Monopolize Rank Scores
A
B
C
D
E
A is a dangling node with no outgoing relationships. Teleportation is used to overcome dead ends.
C, D, and E are circular references with no way out of the group. A dampening factor is used to introduce random node visits.

Figure 5-12. Rank sink is caused by a node, or group of nodes, without outgoing relationships.

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.

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.

When Should I 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.

Example use cases include:

- 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.
- 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".

PageRank with Apache Spark

Now we're ready to execute the PageRank algorithm. GraphFrames supports two implementations of PageRank:

- The first implementation runs PageRank for a fixed number of iterations. This can be run by setting the `maxIter` parameter.
- The second implementation runs PageRank until convergence. This can be run by setting the `tol` parameter.

PageRank with a fixed number of iterations

Let's see an example of the fixed iterations approach:

```
results = g.pageRank(resetProbability=0.15, maxIter=20)
results.vertices.sort("pagerank", ascending=False).show()
```

Notice in Spark that the damping factor is more intuitively called the *reset probability*, with the inverse value. In other words, `resetProbability=0.15` in this example is equivalent to `dampingFactor:0.85` in Neo4j.

If we run that code in pyspark we'll see this output:

id	pageRank
Doug	2.2865372087512252
Mark	2.1424484186137263
Alice	1.520330830262095
Michael	0.7274429252585624
Bridget	0.7274429252585624
Charles	0.5213852310709753
Amy	0.5097143486157744
David	0.36655842368870073
James	0.1981396884803788

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.

The relationships in the graph on which we ran the PageRank algorithm don't have weights, so each relationship is considered equal. Relationship weights are added by specifying a `weight` column in the relationships DataFrame.

PageRank until convergence

And now let's try the convergence implementation that will run PageRank until it closes in on a solution within the set tolerance:

```
results = g.pageRank(resetProbability=0.15, tol=0.01)
results.vertices.sort("pagerank", ascending=False).show()
```

If we run that code in pyspark we'll see this output:

id	pageRank
Doug	2.2233188859989745
Mark	2.090451188336932
Alice	1.5056291439101062
Michael	0.733738785109624
Bridget	0.733738785109624
Amy	0.559446807245026
Charles	0.5338811076334145
David	0.40232326274180685
James	0.21747203391449021

The `PageRank` scores for each person are slightly different than with the fixed number of iterations variant, but as we would expect, their order remains the same.

Although convergence on a perfect solution may sound ideal, in some scenarios PageRank cannot mathematically converge. For larger graphs, PageRank execution may be prohibitively long. A tolerance limit helps set an acceptable range for a converged result, but many choose to use (or combine this approach with) the maximum iteration option instead. The maximum iteration setting will generally provide more performance consistency. Regardless of which option you choose, you may need to test several different limits to find what works for your dataset. Larger graphs typcially require more iterations or smaller tolerance than medium-sized graphs for better accuracy.

PageRank with Neo4j

We can also run PageRank in Neo4j. A call to the following procedure will calculate the PageRank for each of the nodes in our graph:

```
CALL algo.pageRank.stream('User', 'FOLLOWS', {iterations:20, dampingFactor:0.85})
YIELD nodeId, score
RETURN algo.getNodeById(nodeId).id AS page, score
ORDER BY score DESC
```

Running this procedure gives the following result:

page	score
Doug	1.6704119999999998
Mark	1.5610085
Alice	1.1106700000000003
Bridget	0.535373
Michael	0.535373
Amy	0.385875
Charles	0.3844895
David	0.2775
James	0.15000000000000002

As with the Spark example, Doug is the most influential user, and Mark follows closely after as the only user that Doug follows. We can see the importance of the nodes relative to each other in Figure 5-13.

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.

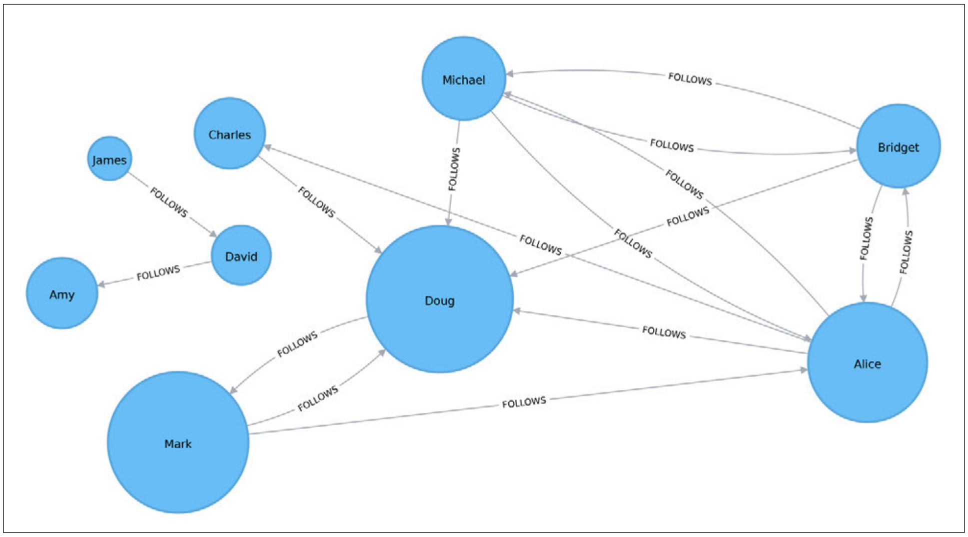


Figure 5-13. Visualization of PageRank

As with our Spark example, the relationships in the graph on which we ran the PageRank algorithm don't have weights, so each relationship is considered equal. Relationship weights can be considered by including the `weightProperty` property in the config passed to the PageRank procedure. For example, if relationships have a property `weight` containing weights, we would pass the following config to the procedure: `weightProperty: "weight"`.

PageRank Variation: Personalized PageRank

Personalized PageRank (PPR) is a variant of the PageRank algorithm that calculates the importance of nodes in a graph from the perspective of a specific node. For PPR, random jumps refer back to a given set of starting nodes. This biases results toward, or personalizes for, the start node. This bias and localization make PPR useful for highly targeted recommendations.

Personalized PageRank with Apache Spark

We can calculate the personalized PageRank score for a given node by passing in the `sourceId` parameter. The following code calculates the PPR for Doug:

```
me = "Doug"
results = g.pageRank(resetProbability=0.15, maxIter=20, sourceId=me)
people_to_follow = results.vertices.sort("pagerank", ascending=False)

already_follows = list(g.edges.filter(f"src = '{me}'").toPandas()["dst"])
people_to_exclude = already_follows + [me]

people_to_follow[~people_to_follow.id.isin(people_to_exclude)].show()
```

The results of this query could be used to make recommendations for people who Doug should follow. Notice that we are also making sure that we exclude people who Doug already follows, as well as himself, from our final result.

If we run that code in pyspark we'll see this output:

id	pageRank
Alice	0.1650183746272782
Michael	0.048842467744891996
Bridget	0.048842467744891996
Charles	0.03497796119878669
David	0.0
James	0.0
Amy	0.0

Alice is the best suggestion for somebody that Doug should follow, but we might suggest Michael and Bridget as well.

Summary

Centrality algorithms are an excellent tool for identifying influencers in a network. In this chapter we've learned about the prototypical centrality algorithms: Degree Centrality, Closeness Centrality, Betweenness Centrality, and PageRank. We've also covered several variations to deal with issues such as long runtimes and isolated components, as well as options for alternative uses.

There are many wide-ranging uses for centrality algorithms, and we encourage their exploration for a variety of analyses. You can apply what we've learned 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.

Next, we'll turn to community detection algorithms that look at groups and partitions.