CHAPTER 6

Community Detection Algorithms

Community formation is common in all types of networks, and identifying them is essential for evaluating group behavior 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 behavior 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'll provide details on the most representative community detection algorithms:

- Triangle Count and Clustering Coefficient for overall relationship density
- Strongly Connected Components and 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'll explain how the algorithms work and show examples in Apache Spark and Neo4j. In cases where an algorithm is only available in one platform, we'll provide just one example. We use weighted relationships for these algorithms because they're typically used to capture the significance of different relationships.

Figure 6-1 gives an overview of the differences between the community detection algorithms covered here, and Table 6-1 provides a quick reference as to what each algorithm calculates with example uses.

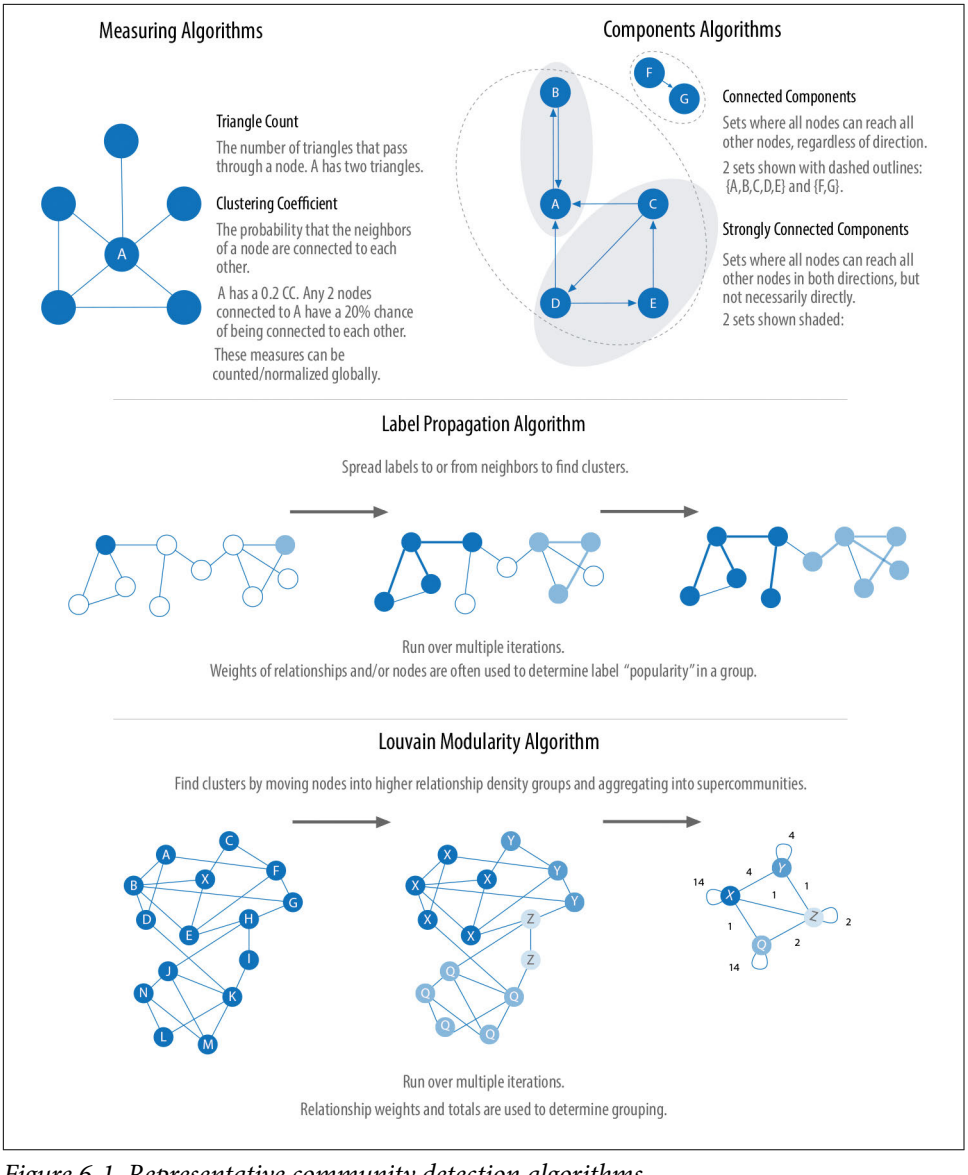

Figure 6-1. Representative community detection algorithms

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. In each section, we use the terms that are most prominent in the literature for a particular algorithm.

Table 6-1. Overview of community detection algorithms

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

First, we'll describe the data for our examples and walk through importing the data into Spark and Neo4j. The algorithms are covered in the order listed in Table 6-1. For each, you'll find a short description and advice on when to use it. Most sections also include guidance on when to use related algorithms. We demonstrate example code using sample data at the end of each algorithm section.

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.

Example Graph Data: The 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. You can download the nodes and files from the book's GitHub repository.

Table 6-2. sw-nodes.csv

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

Table 6-3. sw-relationships.csv

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

Figure 6-2 shows the graph that we want to construct. Looking at this graph, we see that 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.

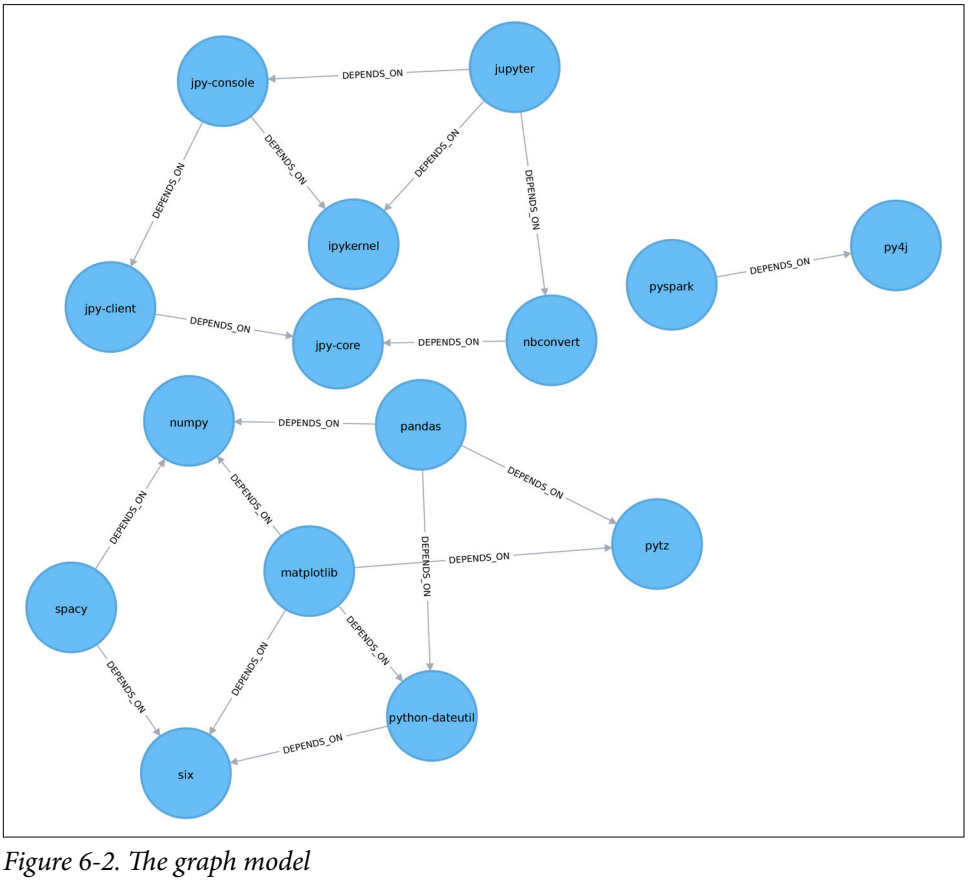

Figure 6-2. The graph model

Let's create graphs in Spark and Neo4j from the example CSV files.

Importing the Data into Apache Spark

We'll first import the packages we need from Apache Spark and the GraphFrames package:

```
from graphframes import *
```

The following function creates a GraphFrame from the example CSV files:

```
def create_software_graph():
    nodes = spark.read.csv("data/sw-nodes.csv", header=True)
    relationships = spark.read.csv("data/sw-relationships.csv", header=True)
    return GraphFrame(nodes, relationships)
```

Now let's call that function:

```
g = create_software_graph()
```

Importing the Data into Neo4j

Next we'll do the same for Neo4j. The following query imports the nodes:

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

And this imports the relationships:

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

Now that we've got our graphs loaded it's on to the 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 behaviors.

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.

Local Clustering Coefficient

The local clustering coefficient of a node is the likelihood that its neighbors 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. Examples of different triangles and clustering coefficients for a node with five relationships are portrayed in Figure 6-3.

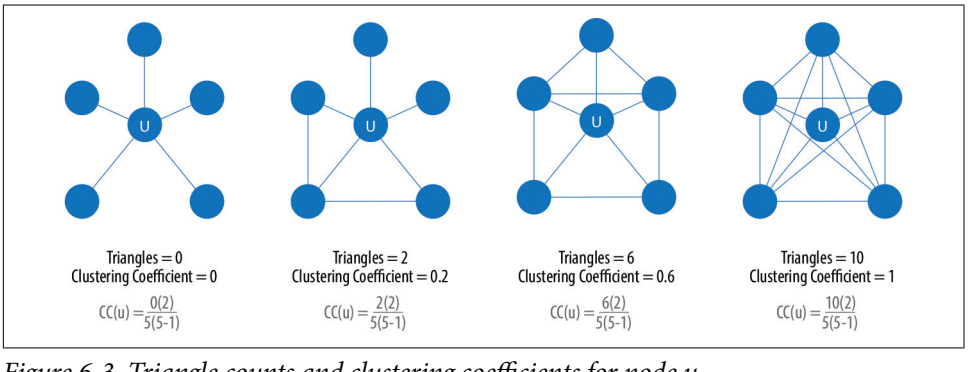

Figure 6-3. Triangle counts and clustering coefficients for node u

Note in Figure 6-3, 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.

The clustering coefficient for a node uses the 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 neighbors of u (this can be obtained by using the number of triangles passing through u).
- $k(u)$ is the degree of u.

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).

When Should I Use Triangle Count and Clustering Coefficient?

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.

Example use cases include:

- 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 neighborhoods 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.
- 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.

Triangle Count with Apache Spark

Now we're ready to execute the Triangle Count algorithm. We can use the following code to do this:

```
result = g.triangleCount()
(result.sort("count", ascending=False)
 .filter('count > 0')
 .show())
```

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

count	id
1	jupyter
1	python-dateutil
1	six
1	ipykernel
1	matplotlib
1	jpy-console

A triangle in this graph would indicate that two of a node's neighbors are also neighbors. Six of our libraries participate in such triangles.

What if we want to know which nodes are in those triangles? That's where a *triangle stream* comes in. For this, we need Neo4j.

Triangles with Neo4j

Getting a stream of the triangles isn't available using Spark, but we can return it using Neo4j:

```
CALL algo.triangle.stream("Library","DEPENDS_ON")
YIELD nodeA, nodeB, nodeC
RETURN algo.getNodeById(nodeA).id AS nodeA,
       algo.getNodeById(nodeB).id AS nodeB,
       algo.getNodeById(nodeC).id AS nodeC
```

Running this procedure gives the following result:

nodeA	nodeB	nodeC
matplotlib	six	python-dateutil
jupyter	jpy-console	ipykernel

We see the same six libraries as we did before, but now we know how they're connected. matplotlib, six, and python-dateutil form one triangle. jupyter, jpy-console, and ipykernel form the other.

We can see these triangles visually in Figure 6-4.

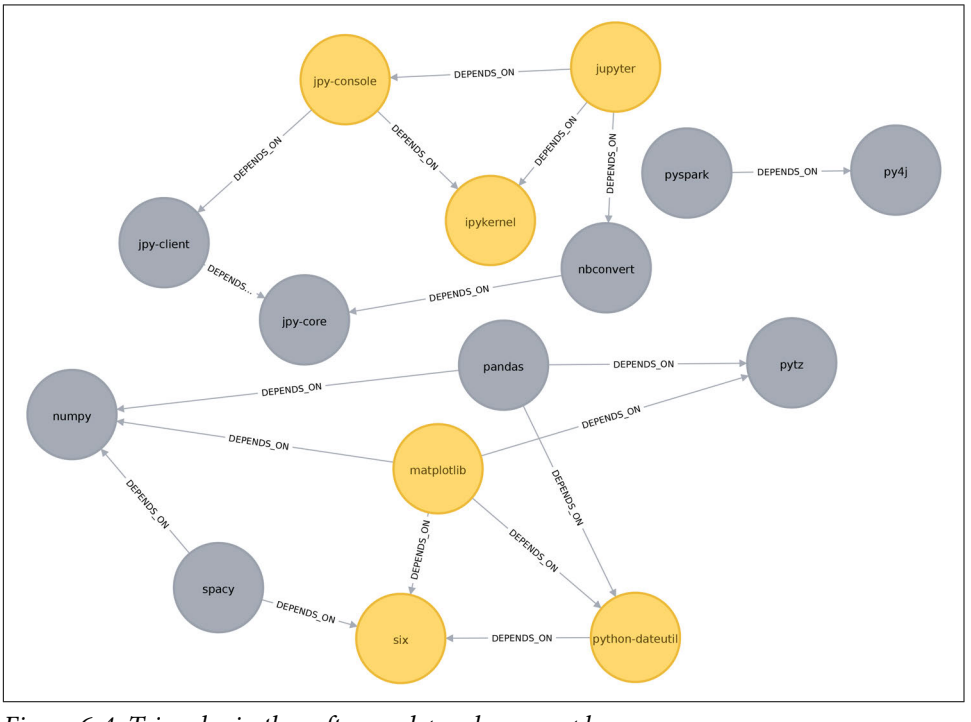

Figure 6-4. Triangles in the software dependency graph

Local Clustering Coefficient with Neo4j

We can also work out the local clustering coefficient. The following query will calculate this for each node:

```
CALL algo.triangleCount.stream('Library', 'DEPENDS_ON')
YIELD nodeId, triangles, coefficient
WHERE coefficient > 0
RETURN algo.getNodeById(nodeId).id AS library, coefficient
ORDER BY coefficient DESC
```

Running this procedure gives the following result:

library	coefficient
ipykernel	1.0
jupyter	0.3333333333333333
jpy-console	0.3333333333333333
six	0.3333333333333333
python-dateutil	0.3333333333333333

library	coefficient
matplotlib	0.16666666666666666

ipykernel has a score of 1, which means that all ipykernel's neighbors are neighbors of each other. We can clearly see that in Figure 6-4. This tells us that the community directly around ipykernel is very cohesive.

We've filtered out nodes with a coefficient score of 0 in this code sample, but nodes with low coefficients may also be interesting. A low score can be an indicator that a node is a *structural hole*—a node that is well connected to nodes in different communities that aren't otherwise connected to each other. This is a method for finding *potential* bridges that we discussed in Chapter 5.

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 6-5 you can see that the nodes in an SCC group don't need to be immediate neighbors, but there must be directional paths between all nodes in the set.

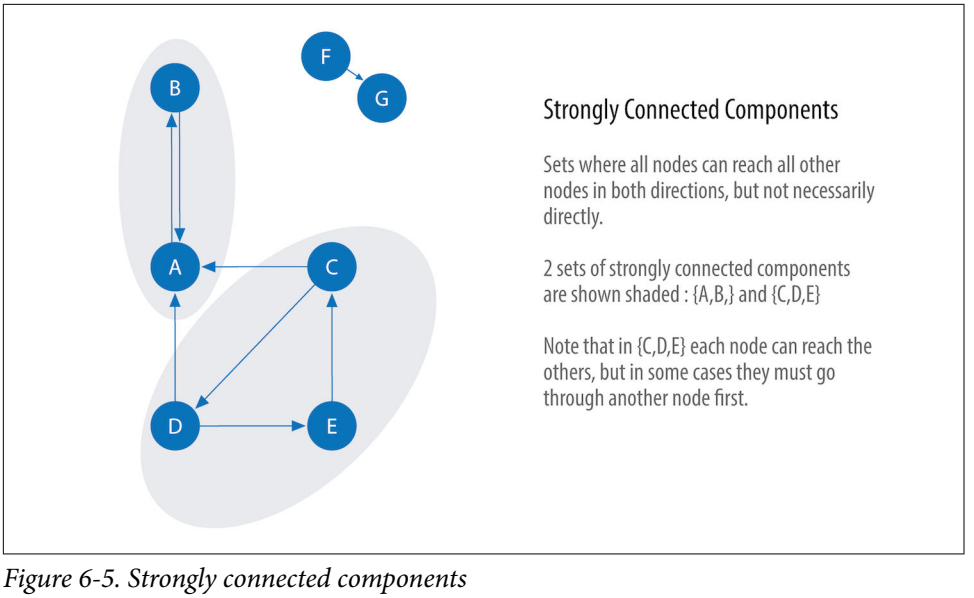

Figure 6-5. Strongly connected components

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.

When Should I 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 behavior 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 intercluster analysis. You can also use SCC to visualize cycles for analyses like finding processes that might deadlock because each subprocess is waiting for another member to take action.

Example use cases include:

- 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 multihop 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.
- 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.

Some algorithms have strategies for escaping infinite loops, but if we're writing our own algorithms or finding nonterminating processes, we can use SCC to check for cycles.

Strongly Connected Components with Apache Spark

Starting with Apache Spark, we'll first import the packages we need from Spark and the GraphFrames package:

```
from graphframes import *
from pyspark.sql import functions as F
```

Now we're ready to execute the Strongly Connected Components algorithm. We'll use it to work out whether there are any circular dependencies in our graph.

Two nodes can only be in the same strongly connected component if there are paths between them in both directions.

We write the following code to do this:

```
result = g.stronglyConnectedComponents(maxIter=10)
(result.sort("component")
 .groupby("component")
 .agg(F.collect_list("id").alias("libraries"))
 .show(truncate=False))
```

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

component	libraries
180388626432	[jpy-core]
223338299392	[spacy]
498216206336	[numpy]
523986010112	[six]
549755813888	[pandas]
558345748480	[nbconvert]
661424963584	[ipykernel]
721554505728	[jupyter]
764504178688	[jpy-client]
833223655424	[pytz]
910533066752	[python-dateutil]
936302870528	[pyspark]
944892805120	[matplotlib]
1099511627776	[jpy-console]
1279900254208	[py4j]

You might notice that every library node is assigned to a unique component. This is the partition or subgroup it belongs to, and as we (hopefully!) expected, every node is in its own partition. This means our software project has no circular dependencies amongst these libraries.

Strongly Connected Components with Neo4j

Let's run the same algorithm using Neo4j. Execute the following query to run the algorithm:

```
CALL algo.scc.stream("Library", "DEPENDS_ON")
YIELD nodeId, partition
RETURN partition, collect(algo.getNodeById(nodeId)) AS libraries
ORDER BY size(libraries) DESC
```

The parameters passed to this algorithm are:

`Library`
: The node label to load from the graph

`DEPENDS_ON`
: The relationship type to load from the graph

This is the output we'll see when we run the query:

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]

As with the Spark example, every node is in its own partition.

So far the algorithm has only revealed that our Python libraries are very well behaved, 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.

The following query adds an extra library that creates a circular dependency between py4j and pyspark:

```
MATCH (py4j:Library {id: "py4j"})
MATCH (pyspark:Library {id: "pyspark"})
MERGE (extra:Library {id: "extra"})
MERGE (py4j)-[:DEPENDS_ON]->(extra)
MERGE (extra)-[:DEPENDS_ON]->(pyspark)
```

We can clearly see the circular dependency that got created in Figure 6-6.

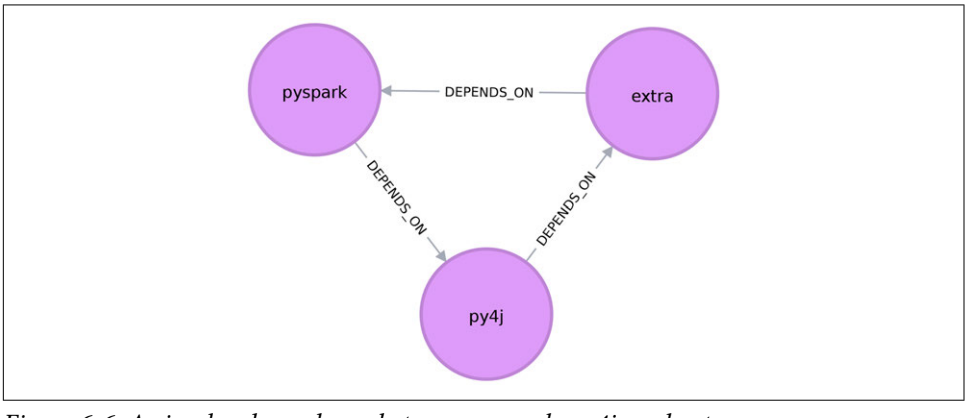

Figure 6-6. A circular dependency between pyspark, py4j, and extra

Now if we run the SCC algorithm again we'll see a slightly different result:

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]

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

Before we move on to the next algorithm we'll delete the extra library and its relationships from the graph:

```
MATCH (extra:Library {id: "extra"})
DETACH DELETE extra
```

Connected Components

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".

When Should I 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.

Example use cases include:

- 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.

Connected Components with Apache Spark

Starting with Apache Spark, we'll first import the packages we need from Spark and the GraphFrames package:

```
from pyspark.sql import functions as F
```

Now we're ready to execute the Connected Components algorithm.

Two nodes can be in the same connected component if there is a path between them in either direction.

We write the following code to do this:

```
result = g.connectedComponents()
(result.sort("component")
 .groupby("component")
 .agg(F.collect_list("id").alias("libraries"))
 .show(truncate=False))
```

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

component	libraries
180388626432	[jpy-core, nbconvert, ipykernel, jupyter, jpy-client, jpy-console]
223338299392	[spacy, numpy, six, pandas, pytz, python-dateutil, matplotlib]
936302870528	[pyspark, py4j]

The results show three clusters of nodes, which can also be seen in Figure 6-7.

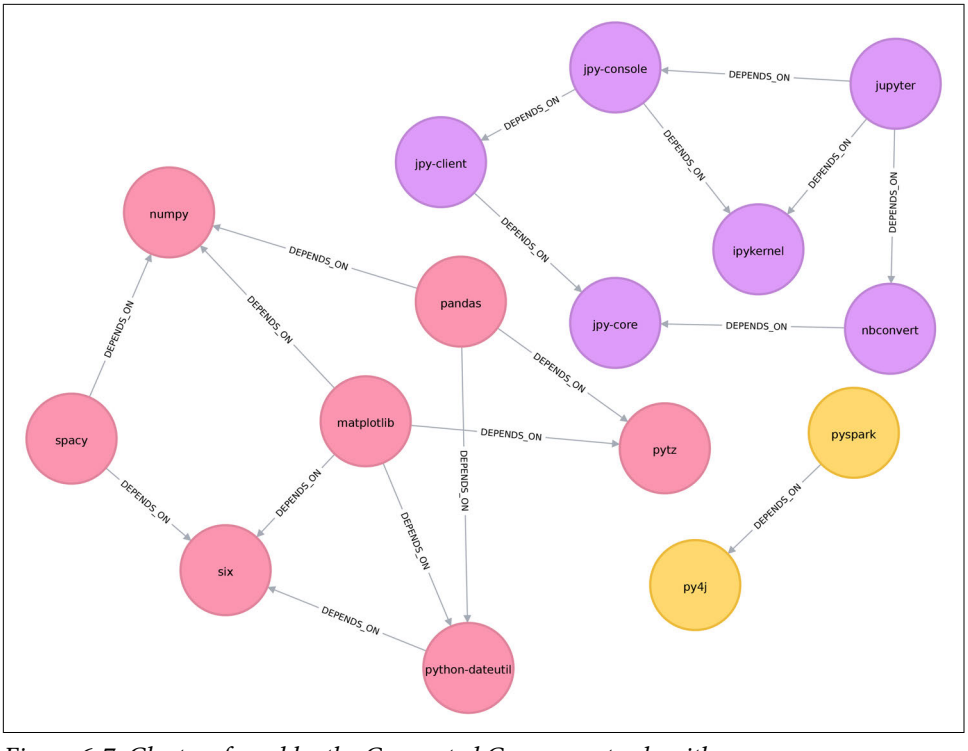

Figure 6-7. Clusters found by the Connected Components algorithm

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.

Connected Components with Neo4j

We can also execute this algorithm in Neo4j by running the following query:

```
CALL algo.unionFind.stream("Library", "DEPENDS_ON")
YIELD nodeId,setId
RETURN setId, collect(algo.getNodeById(nodeId)) AS libraries
ORDER BY size(libraries) DESC
```

The parameters passed to this algorithm are:

`Library`
: The node label to load from the graph

`DEPENDS_ON`
: The relationship type to load from the graph

Here's the output:

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

As expected, we get exactly the same results as we did with Spark.

Both of the 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.

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 neighbors. 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 semisupervised learning because you can seed the process with preassigned, indicative node labels.

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 neighborhood 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".

Figure 6-8 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.

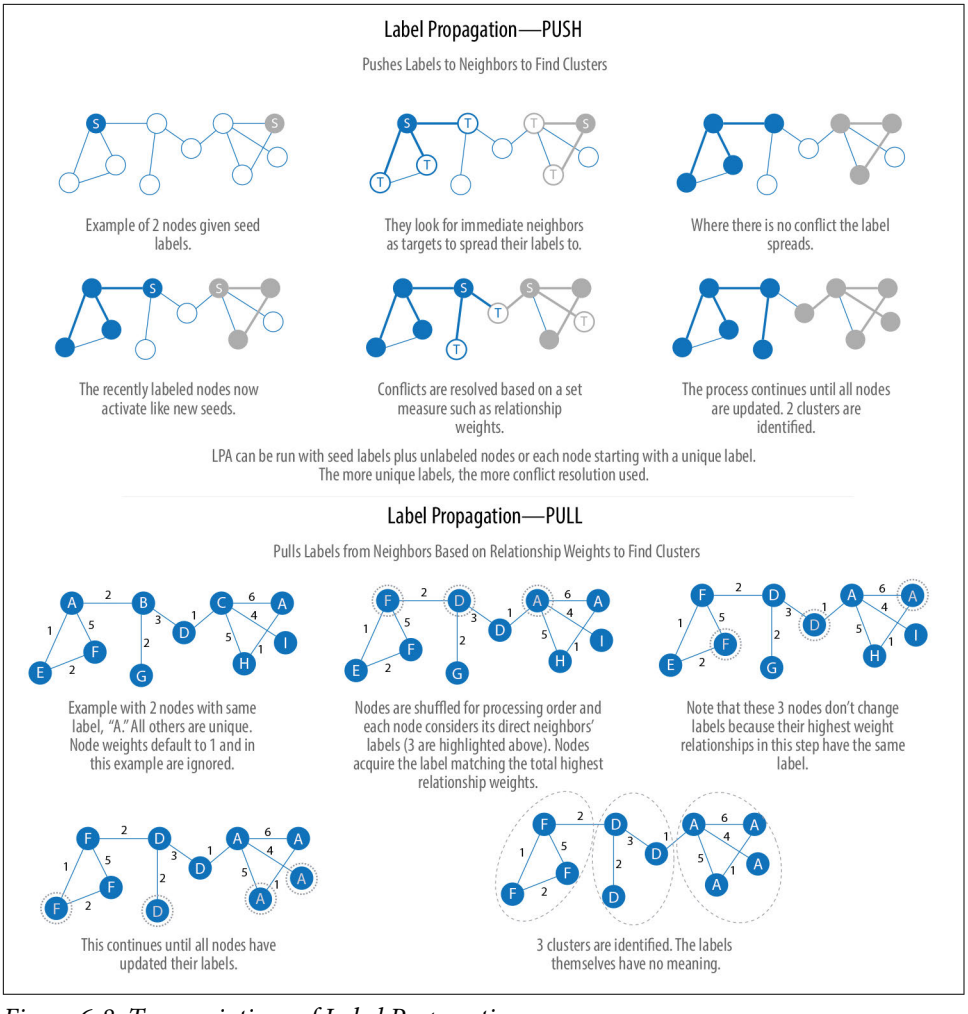

Figure 6-8. Two variations of Label Propagation

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 neighbor nodes *and* their relationships. Ties are broken uniformly and randomly.
4. LPA reaches convergence when each node has the majority label of its neighbors.

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.

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 unlabeled. Unlabeled nodes are more likely to adopt the preliminary 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 labeled data, along with a larger amount of unlabeled data. We can also run the algorithm repeatedly on graphs as they evolve.

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. Spark and 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.

When Should I 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.

Example use cases include:

- 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.

- 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.

Label Propagation with Apache Spark

Starting with Apache Spark, we'll first import the packages we need from Spark and the GraphFrames package:

```
from pyspark.sql import functions as F
```

Now we're ready to execute the Label Propagation algorithm. We write the following code to do this:

```
result = g.labelPropagation(maxIter=10)
(result
.sort("label")
.groupby("label")
.agg(F.collect_list("id"))
.show(truncate=False))
```

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

label	collect_list(id)
180388626432	[jpy-core, jpy-console, jupyter]
223338299392	[matplotlib, spacy]
498216206336	[python-dateutil, numpy, six, pytz]
549755813888	[pandas]
558345748480	[nbconvert, ipykernel, jpy-client]
936302870528	[pyspark]
1279900254208	[py4j]

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 neighbors (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 neighbors 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.

Label Propagation with Neo4j

Now let's try the same algorithm with Neo4j. We can execute LPA by running the following query:

```
CALL algo.labelPropagation.stream("Library", "DEPENDS_ON",
  { iterations: 10 })
YIELD nodeId, label
RETURN label,
       collect(algo.getNodeById(nodeId).id) AS libraries
ORDER BY size(libraries) DESC
```

The parameters passed to this algorithm are:

`Library`
: The node label to load from the graph

`DEPENDS_ON`
: The relationship type to load from the graph

`iterations: 10`
: The maximum number of iterations to run

These are the results we'd see:

label	libraries
11	[matplotlib, spacy, six, pandas, python-dateutil]
10	[jupyter, jpy-console, nbconvert, jpy-client, jpy-core]
4	[pyspark, py4j]
8	[ipykernel]
13	[numpy]
0	[pytz]

The results, which can also be seen visually in Figure 6-9, are fairly similar to those we got with Apache Spark.

Figure 6-9. Clusters found by the Label Propagation algorithm

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.

To do this, we pass the `DIRECTION:BOTH` parameter to the algorithm:

```
CALL algo.labelPropagation.stream("Library", "DEPENDS_ON",
  { iterations: 10, direction: "BOTH" })
YIELD nodeId, label
RETURN label,
       collect(algo.getNodeById(nodeId).id) AS libraries
ORDER BY size(libraries) DESC
```

If we run that, we'll get the following output:

label	libraries
11	[pytz, matplotlib, spacy, six, pandas, numpy, python-dateutil]
10	[nbconvert, jpy-client, jpy-core]
6	[jupyter, jpy-console, ipykernel]
4	[pyspark, py4j]

The number of clusters has reduced from six to four, and all the nodes in the matplotlib part of the graph are now grouped together. This can be seen more clearly in Figure 6-10.

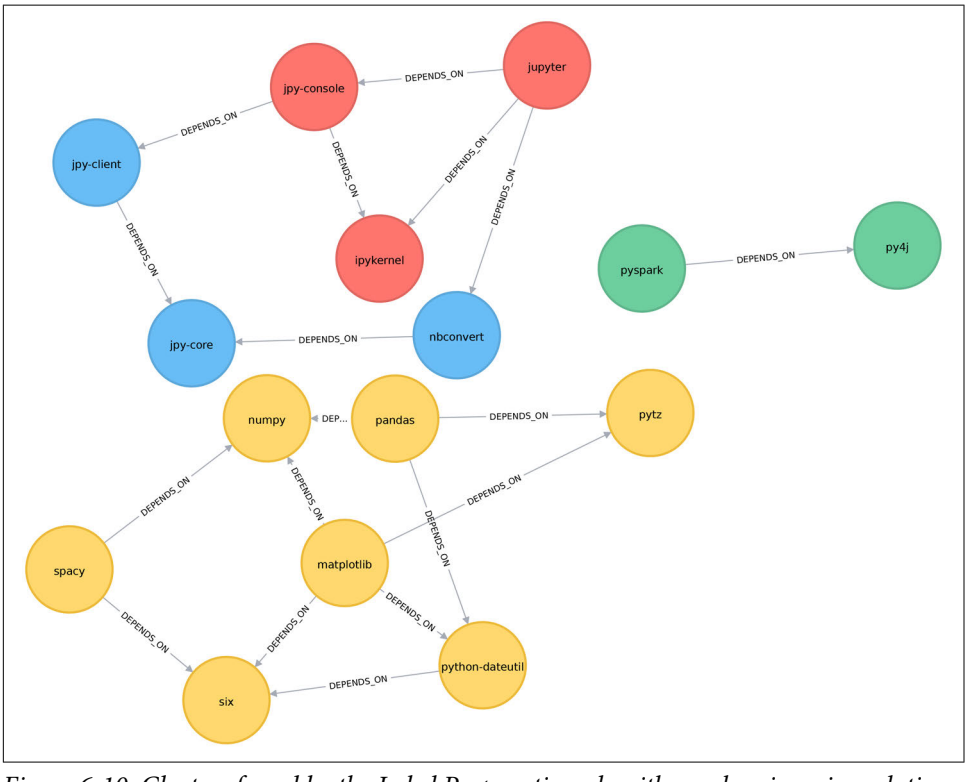

Figure 6-10. Clusters found by the Label Propagation algorithm, when ignoring relationship direction

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.

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*.

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.

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.

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 6-11 illustrates several different modularity scores based on node groupings.

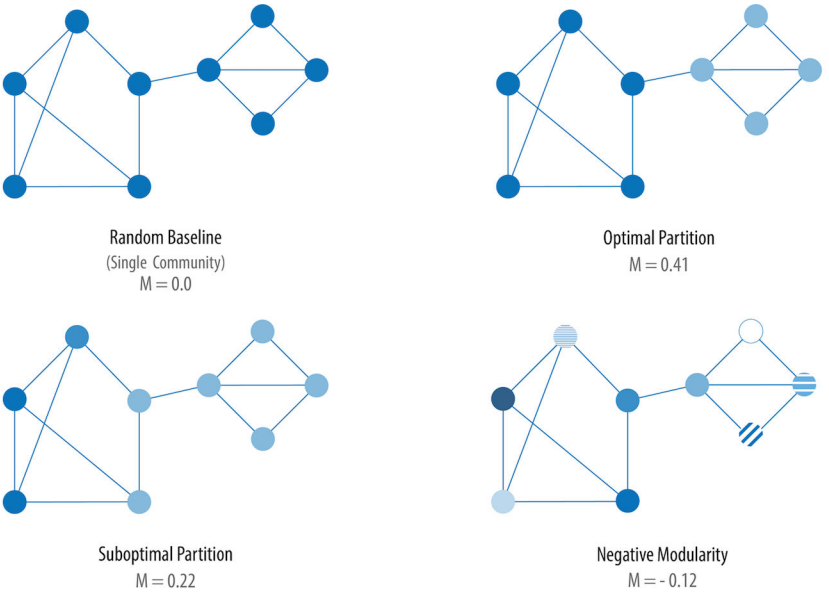

Figure 6-11. Four modularity scores based on different partitioning choices

The formula for the modularity of a group is:

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

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.

The calculation for the optimal partition at the top of Figure 6-11 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 $M = 0.205 + 0.206 = 0.41$

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.

The algorithm consists of repeated application of two steps, as illustrated in Figure 6-12.

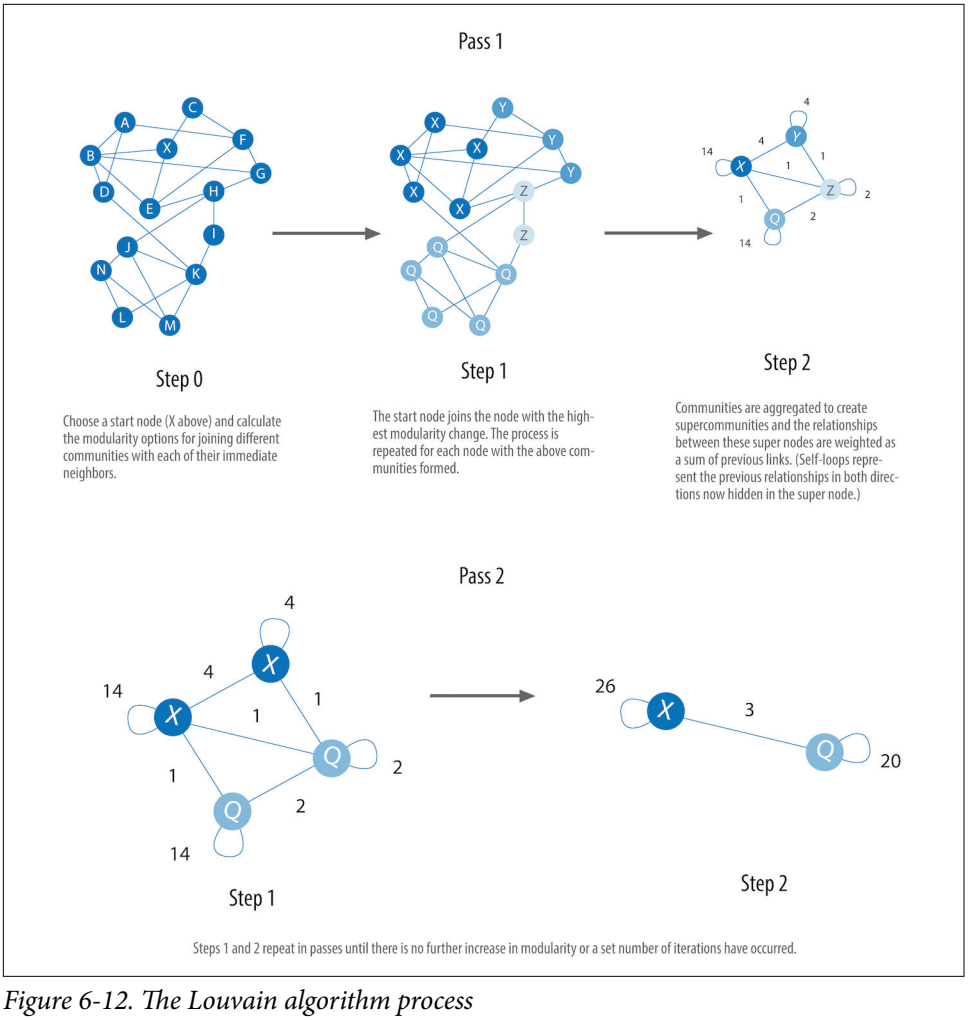

Figure 6-12. The Louvain algorithm process

The Louvain algorithm's steps include:

1. A "greedy" assignment of nodes to communities, favoring 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.

Part of the first optimization step is evaluating the modularity of a group. Louvain uses the following formula to accomplish this:

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

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.

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.

When Should I 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 where 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.

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 modeling 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.

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, we recommend using any modularity algorithm as a guide for gross estimation but not complete accuracy.

Louvain with Neo4j

Let's see the Louvain algorithm in action. We can execute the following query to run the algorithm over our graph:

```
CALL algo.louvain.stream("Library", "DEPENDS_ON")
YIELD nodeId, communities
RETURN algo.getNodeById(nodeId).id AS libraries, communities
```

The parameters passed to this algorithm are:

`Library`
: The node label to load from the graph

`DEPENDS_ON`
: The relationship type to load from the graph

These are the results:

libraries	communities
pytz	[0, 0]

libraries	communities
pyspark	[1, 1]
matplotlib	[2, 0]
spacy	[2, 0]
py4j	[1, 1]
jupyter	[3, 2]
jpy-console	[3, 2]
nbconvert	[4, 2]
ipykernel	[3, 2]
jpy-client	[4, 2]
jpy-core	[4, 2]
six	[2, 0]
pandas	[0, 0]
numpy	[2, 0]
python-dateutil	[2, 0]

The `communities` 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 `[2,0]`. This means that matplotlib's final community is labeled `0` and its intermediate community is labeled `2`.

It's easier to see how this works if we store these communities using the write version of the algorithm and then query it afterwards. The following query will run the Louvain algorithm and store the result in the `communities` property on each node:

```
CALL algo.louvain("Library", "DEPENDS_ON")
```

We could also store the resulting communities using the streaming version of the algorithm, followed by calling the `SET` clause to store the result. The following query shows how we could do this:

```
CALL algo.louvain.stream("Library", "DEPENDS_ON")
YIELD nodeId, communities
WITH algo.getNodeById(nodeId) AS node, communities
SET node.communities = communities
```

Once we've run either of those queries, we can write the following query to find the final clusters:

```
MATCH (l:Library)
RETURN l.communities[-1] AS community, collect(l.id) AS libraries
ORDER BY size(libraries) DESC
```

`l.communities[-1]` returns the last item from the underlying array that this property stores.

Running the query yields this output:

community	libraries
0	[pytz, matplotlib, spacy, six, pandas, numpy, python-dateutil]
2	[jupyter, jpy-console, nbconvert, ipykernel, jpy-client, jpy-core]
1	[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 python-dateutil. We can see this more clearly in Figure 6-13.

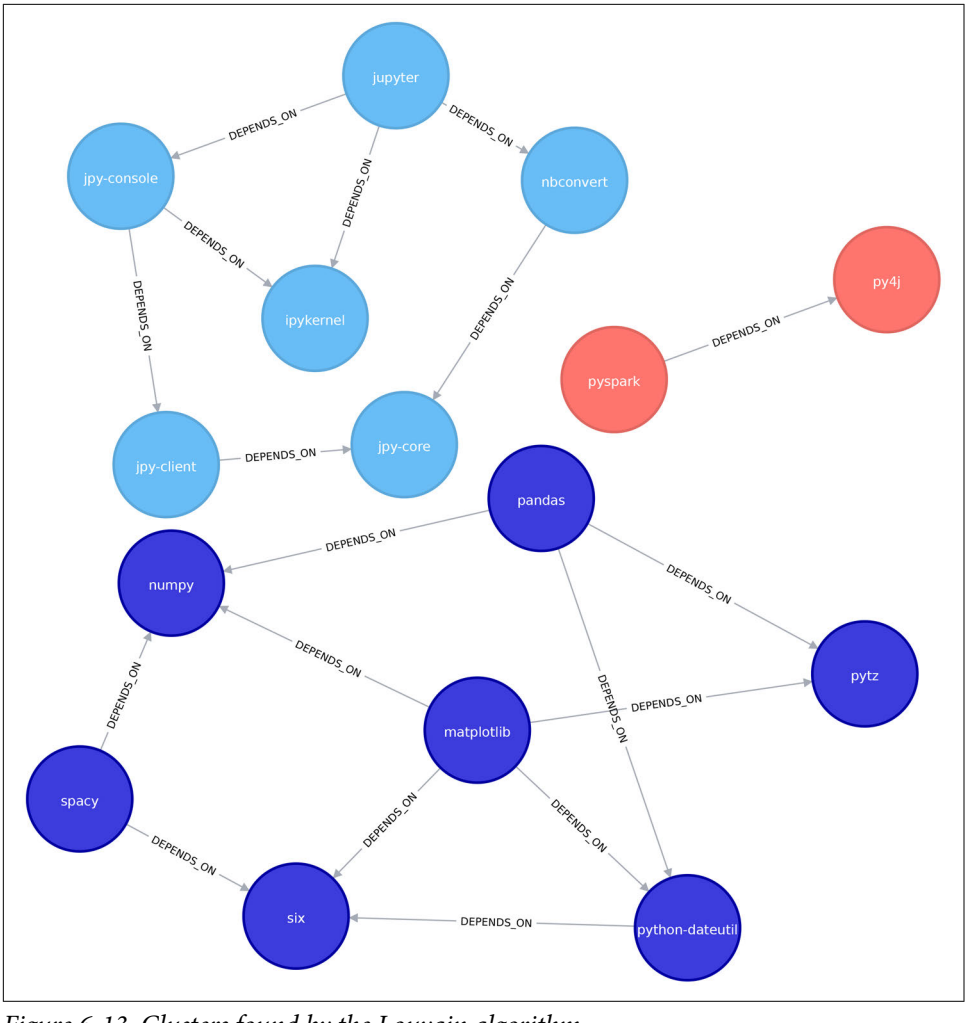

Figure 6-13. Clusters found by the Louvain algorithm

An additional feature of the Louvain algorithm is that we can see the intermediate clustering as well. This will show us finer-grained clusters than the final layer did:

```
MATCH (l:Library)
RETURN l.communities[0] AS community, collect(l.id) AS libraries
ORDER BY size(libraries) DESC
```

Running that query gives this output:

community	libraries
2	[matplotlib, spacy, six, python-dateutil]
4	[nbconvert, jpy-client, jpy-core]

community	libraries
3	[jupyter, jpy-console, ipykernel]
1	[pyspark, py4j]
0	[pytz, pandas]
5	[numpy]

The libraries in the matplotlib community have now broken down into three smaller communities:

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

We can see this breakdown visually in Figure 6-14.

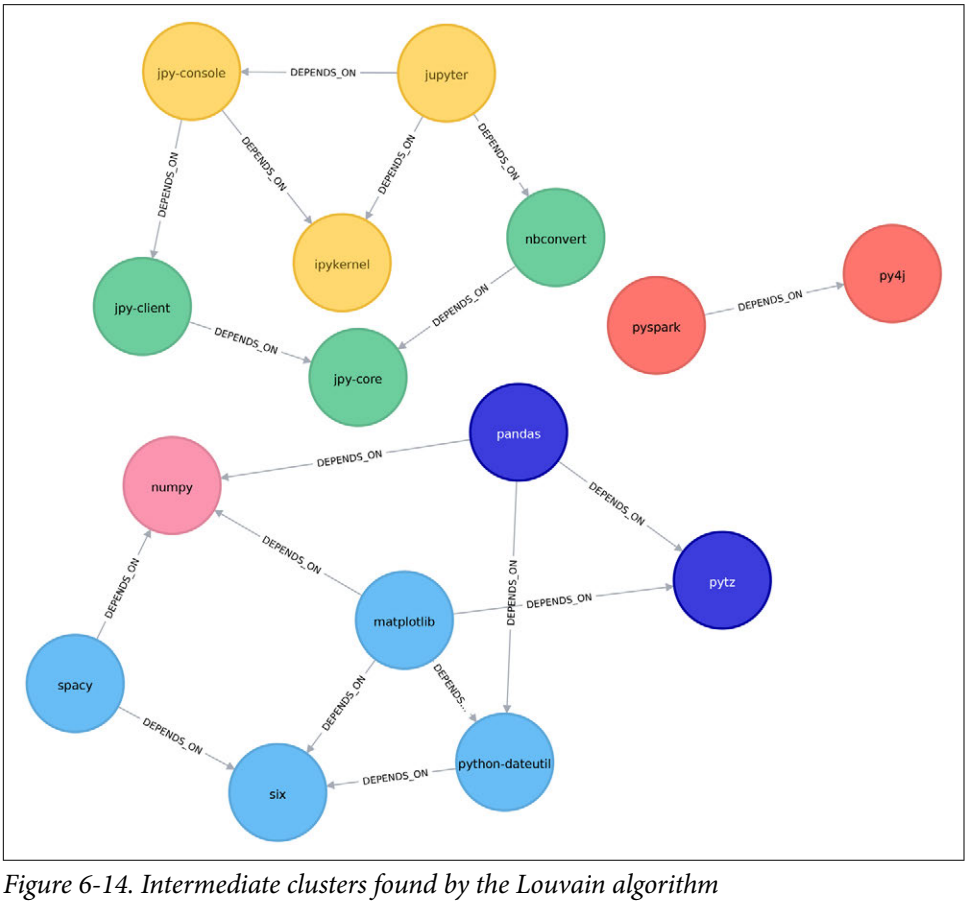

Figure 6-14. Intermediate clusters found by the Louvain algorithm

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.

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.

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 where 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.

Summary

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

In this chapter, we started by learning about 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.

We finished with Label Propagation and Louvain, two nondeterministic algorithms which are better able to detect finer-grained communities. Louvain also showed us a hierarchy of communities at different scales.

In the next chapter, we'll take a much larger dataset and learn how to combine the algorithms together to gain even more insight into our connected data.