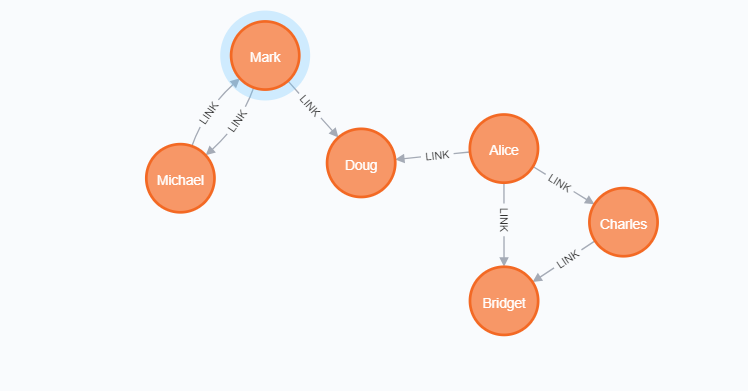
In this section we will show examples of running the Louvain community detection algorithm on a concrete graph. The intention is to illustrate what the results look like and to provide a guide in how to make use of the algorithm in a real setting. We will do this on a small social network graph of a handful nodes connected in a particular pattern. The example graph looks like this:



**The following Cypher statement will create the example graph in the Neo4j database:**

CREATE

(nAlice:User {name: 'Alice', seed: 42}),

(nBridget:User {name: 'Bridget', seed: 42}),

(nCharles:User {name: 'Charles', seed: 42}),

(nDoug:User {name: 'Doug'}),

(nMark:User {name: 'Mark'}),

(nMichael:User {name: 'Michael'}),

(nAlice)-[:LINK {weight: 1}]->(nBridget),

(nAlice)-[:LINK {weight: 1}]->(nCharles),

(nCharles)-[:LINK {weight: 1}]->(nBridget),

(nAlice)-[:LINK {weight: 5}]->(nDoug),

(nMark)-[:LINK {weight: 1}]->(nDoug),

(nMark)-[:LINK {weight: 1}]->(nMichael),

(nMichael)-[:LINK {weight: 1}]->(nMark);

This graph has two clusters of *Users*, that are closely connected. Between those clusters there is one single edge. The relationships that connect the nodes in each component have a property weight which determines the strength of the relationship.

We can now create the graph and store it in the graph catalog. We load the LINK relationships with orientation set to UNDIRECTED as this works best with the Louvain algorithm.

**The following statement will create the graph and store it in the graph catalog.**

CALL gds.graph.create(

'myGraph',

'User',

{

LINK: {

orientation: 'UNDIRECTED'

}

},

{

nodeProperties: 'seed',

relationshipProperties: 'weight'

}

)

In the following examples we will demonstrate using the Louvain algorithm on this graph.



Memory Estimation

First off, we will estimate the cost of running the algorithm using the estimate procedure. This can be done with any execution mode. We will use the write mode in this example. Estimating the algorithm is useful to understand the memory impact that running the algorithm on your graph will have. When you later actually run the algorithm in one of the execution modes the system will perform an estimation. If the estimation shows that there is a very high probability of the execution going over its memory limitations, the execution is prohibited. To read more about this, see [Section 3.1.3, “Automatic estimation and execution blocking”](https://neo4j.com/docs/graph-data-science/current/common-usage/memory-estimation/#estimate-heap-control).

For more details on estimate in general, see [Section 3.1, “Memory Estimation”](https://neo4j.com/docs/graph-data-science/current/common-usage/memory-estimation/).

**The following will estimate the memory requirements for running the algorithm:**

CALL gds.louvain.write.estimate('myGraph', { writeProperty: 'community' })

YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory

| **nodeCount** | **relationshipCount** | **bytesMin** | **bytesMax** | **requiredMemory** |
| --- | --- | --- | --- | --- |
| 6 | 14 | 5321 | 580112 | "[5321 Bytes ... 566 KiB]" |

Stream

In the stream execution mode, the algorithm returns the community ID for each node. This allows us to inspect the results directly or post-process them in Cypher without any side effects. For example, we can order the results to find the nodes with the highest betweenness centrality.

For more details on the stream mode in general, see [Section 3.3.1, “Stream”](https://neo4j.com/docs/graph-data-science/current/common-usage/running-algos/#running-algos-stream).

**The following will run the algorithm and stream results:**

CALL gds.louvain.stream('myGraph')

YIELD nodeId, communityId, intermediateCommunityIds

RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds

ORDER BY name ASC

| **name** | **communityId** | **intermediateCommunityIds** |
| --- | --- | --- |
| "Alice" | 2 | null |
| "Bridget" | 2 | null |
| "Charles" | 2 | null |
| "Doug" | 5 | null |
| "Mark" | 5 | null |
| "Michael" | 5 | null |

We use default values for the procedure configuration parameter. Levels and innerIterations are set to 10 and the tolerance value is 0.0001. Because we did not set the value of includeIntermediateCommunities to true, the column communities is always null.

Stats

In the stats execution mode, the algorithm returns a single row containing a summary of the algorithm result. In particular, Betweenness Centrality returns the minimum, maximum and sum of all centrality scores. This execution mode does not have any side effects. It can be useful for evaluating algorithm performance by inspecting the computeMillis return item. In the examples below we will omit returning the timings. The full signature of the procedure can be found in [the syntax section](https://neo4j.com/docs/graph-data-science/current/algorithms/louvain/#algorithms-louvain-syntax).

For more details on the stats mode in general, see [Section 3.3.2, “Stats”](https://neo4j.com/docs/graph-data-science/current/common-usage/running-algos/#running-algos-stats).

**The following will run the algorithm and returns the result in form of statistical and measurement values.**

CALL gds.louvain.stats('myGraph')

YIELD communityCount

| Table 6.106. Results |
| --- |
| **communityCount** |
| 2 |

Mutate

The mutate execution mode extends the stats mode with an important side effect: updating the named graph with a new node property containing the community ID for that node. The name of the new property is specified using the mandatory configuration parameter mutateProperty. The result is a single summary row, similar to stats, but with some additional metrics. The mutate mode is especially useful when multiple algorithms are used in conjunction.

For more details on the mutate mode in general, see [Section 3.3.3, “Mutate”](https://neo4j.com/docs/graph-data-science/current/common-usage/running-algos/#running-algos-mutate).

**The following will run the algorithm and store the results in myGraph:**

CALL gds.louvain.mutate('myGraph', { mutateProperty: 'communityId' })

YIELD communityCount, modularity, modularities

| **communityCount** | **modularity** | **modularities** |
| --- | --- | --- |
| 2 | 0.3571428571428571 | [0.3571428571428571] |

In mutate mode, only a single row is returned by the procedure. The result contains meta information, like the number of identified communities and the modularity values. In contrast to the write mode the result is written to the GDS in-memory graph instead of the Neo4j database.

Write

The write execution mode extends the stats mode with an important side effect: writing the community ID for each node as a property to the Neo4j database. The name of the new property is specified using the mandatory configuration parameter writeProperty. The result is a single summary row, similar to stats, but with some additional metrics. The write mode enables directly persisting the results to the database.

For more details on the write mode in general, see [Section 3.3.4, “Write”](https://neo4j.com/docs/graph-data-science/current/common-usage/running-algos/#running-algos-write).

**The following run the algorithm, and write back results:**

CALL gds.louvain.write('myGraph', { writeProperty: 'community' })

YIELD communityCount, modularity, modularities

| **communityCount** | **modularity** | **modularities** |
| --- | --- | --- |
| 2 | 0.3571428571428571 | [0.3571428571428571] |

When writing back the results, only a single row is returned by the procedure. The result contains meta information, like the number of identified communities and the modularity values.

Weighted

The Louvain algorithm can also run on weighted graphs, taking the given relationship weights into concern when calculating the modularity.

**The following will run the algorithm on a weighted graph and stream results:**

CALL gds.louvain.stream('myGraph', { relationshipWeightProperty: 'weight' })

YIELD nodeId, communityId, intermediateCommunityIds

RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds

ORDER BY name ASC

| **name** | **communityId** | **intermediateCommunityIds** |
| --- | --- | --- |
| "Alice" | 3 | null |
| "Bridget" | 2 | null |
| "Charles" | 2 | null |
| "Doug" | 3 | null |
| "Mark" | 5 | null |
| "Michael" | 5 | null |

Using the weighted relationships, we see that Alice and Doug have formed their own community, as their link is much stronger than all the others.

Seeded

The Louvain algorithm can be run incrementally, by providing a seed property. With the seed property an initial community mapping can be supplied for a subset of the loaded nodes. The algorithm will try to keep the seeded community IDs.

**The following will run the algorithm and stream results:**

CALL gds.louvain.stream('myGraph', { seedProperty: 'seed' })

YIELD nodeId, communityId, intermediateCommunityIds

RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds

ORDER BY name ASC

| **name** | **communityId** | **intermediateCommunityIds** |
| --- | --- | --- |
| "Alice" | 42 | null |
| "Bridget" | 42 | null |
| "Charles" | 42 | null |
| "Doug" | 47 | null |
| "Mark" | 47 | null |
| "Michael" | 47 | null |

Using the seeded graph, we see that the community around Alice keeps its initial community ID of 42. The other community is assigned a new community ID, which is guaranteed to be larger than the largest seeded community ID. Note that the consecutiveIds configuration option cannot be used in combination with seeding in order to retain the seeding values.

Stream intermediate communities

As described before, Louvain is a hierarchical clustering algorithm. That means that after every clustering step all nodes that belong to the same cluster are reduced to a single node. Relationships between nodes of the same cluster become self-relationships, relationships to nodes of other clusters connect to the clusters representative. This condensed graph is then used to run the next level of clustering. The process is repeated until the clusters are stable.

In order to demonstrate this iterative behavior, we need to construct a more complex graph.

CREATE (a:Node {name: 'a'})

CREATE (b:Node {name: 'b'})

CREATE (c:Node {name: 'c'})

CREATE (d:Node {name: 'd'})

CREATE (e:Node {name: 'e'})

CREATE (f:Node {name: 'f'})

CREATE (g:Node {name: 'g'})

CREATE (h:Node {name: 'h'})

CREATE (i:Node {name: 'i'})

CREATE (j:Node {name: 'j'})

CREATE (k:Node {name: 'k'})

CREATE (l:Node {name: 'l'})

CREATE (m:Node {name: 'm'})

CREATE (n:Node {name: 'n'})

CREATE (x:Node {name: 'x'})

CREATE (a)-[:TYPE]->(b)

CREATE (a)-[:TYPE]->(d)

CREATE (a)-[:TYPE]->(f)

CREATE (b)-[:TYPE]->(d)

CREATE (b)-[:TYPE]->(x)

CREATE (b)-[:TYPE]->(g)

CREATE (b)-[:TYPE]->(e)

CREATE (c)-[:TYPE]->(x)

CREATE (c)-[:TYPE]->(f)

CREATE (d)-[:TYPE]->(k)

CREATE (e)-[:TYPE]->(x)

CREATE (e)-[:TYPE]->(f)

CREATE (e)-[:TYPE]->(h)

CREATE (f)-[:TYPE]->(g)

CREATE (g)-[:TYPE]->(h)

CREATE (h)-[:TYPE]->(i)

CREATE (h)-[:TYPE]->(j)

CREATE (i)-[:TYPE]->(k)

CREATE (j)-[:TYPE]->(k)

CREATE (j)-[:TYPE]->(m)

CREATE (j)-[:TYPE]->(n)

CREATE (k)-[:TYPE]->(m)

CREATE (k)-[:TYPE]->(l)

CREATE (l)-[:TYPE]->(n)

CREATE (m)-[:TYPE]->(n);

**The following will load the example graph, run the algorithm and stream results including the intermediate communities:**

CALL gds.louvain.stream({

nodeProjection: 'Node',

relationshipProjection: {

TYPE: {

type: 'TYPE',

orientation: 'undirected',

aggregation: 'NONE'

}

},

includeIntermediateCommunities: true

}) YIELD nodeId, communityId, intermediateCommunityIds

RETURN gds.util.asNode(nodeId).name AS name, communityId, intermediateCommunityIds

ORDER BY name ASC

| **name** | **communityId** | **intermediateCommunityIds** |
| --- | --- | --- |
| "a" | 14 | [3, 14] |
| "b" | 14 | [3, 14] |
| "c" | 14 | [14, 14] |
| "d" | 14 | [3, 14] |
| "e" | 14 | [14, 14] |
| "f" | 14 | [14, 14] |
| "g" | 7 | [7, 7] |
| "h" | 7 | [7, 7] |
| "i" | 7 | [7, 7] |
| "j" | 12 | [12, 12] |
| "k" | 12 | [12, 12] |
| "l" | 12 | [12, 12] |
| "m" | 12 | [12, 12] |
| "n" | 12 | [12, 12] |
| "x" | 14 | [14, 14] |

In this example graph, after the first iteration we see 4 clusters, which in the second iteration are reduced to three.