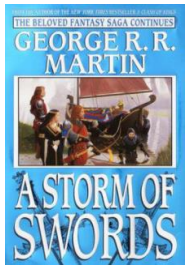
Graph Algorithms and Text - Game of Thrones Dataset



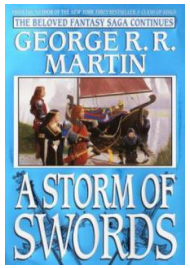
The example dataset used to demonstrate the GDS library is based on the Game of Thrones fantasy saga.

The dataset is partly based on the following works:
Network of Thrones, A Song of Math and Westeros, research by Dr. Andrew Beveridge.

A. Beveridge and J. Shan, "Network of Thrones," Math Horizons Magazine , Vol. 23, No. 4 (2016), pp. 18-22Game of Thrones, Explore deaths and battles from this fantasy world, by Myles O'Neill

https://www.kaggle.com/Game of Thrones, by Tomaz Bratanic, GitHub repository.

Graph Algorithms and Text - Game of Thrones Dataset



The graph contains

:Person nodes, representing the characters, and

:INTERACTS relationships, representing the characters' interactions.

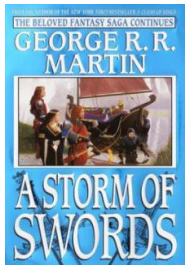
An interaction occurs each time two characters' names (or nicknames) **appear within 15 words of one another** in the book text.

The (:Person)-[:INTERACTS]→(:Person) graph is enriched with data on houses, battles, commanders, kings, knights, regions, locations, and deaths.

For more information about the data extraction process, see *Network of Thrones, A Song of Math and Westeros, research by Dr. Andrew Beveridge.*

Graph Algorithms and Text - Game of Thrones

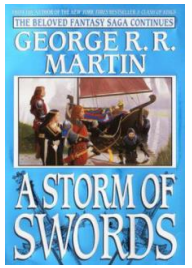
Importing the Data - constraints




```
:config "enableMultiStatementMode":true

CREATE CONSTRAINT ON (n:Location) ASSERT n.name IS
UNIQUE;
CREATE CONSTRAINT ON (n:Region) ASSERT n.name IS UNIQUE;
CREATE CONSTRAINT ON (n:Battle) ASSERT n.name IS UNIQUE;
CREATE CONSTRAINT ON (n:Person) ASSERT n.name IS UNIQUE;
CREATE CONSTRAINT ON (n:House) ASSERT n.name IS UNIQUE;
```

Graph Algorithms and Text - Game of Thrones

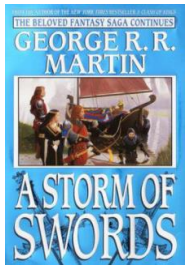
Importing the Data – battles.csv



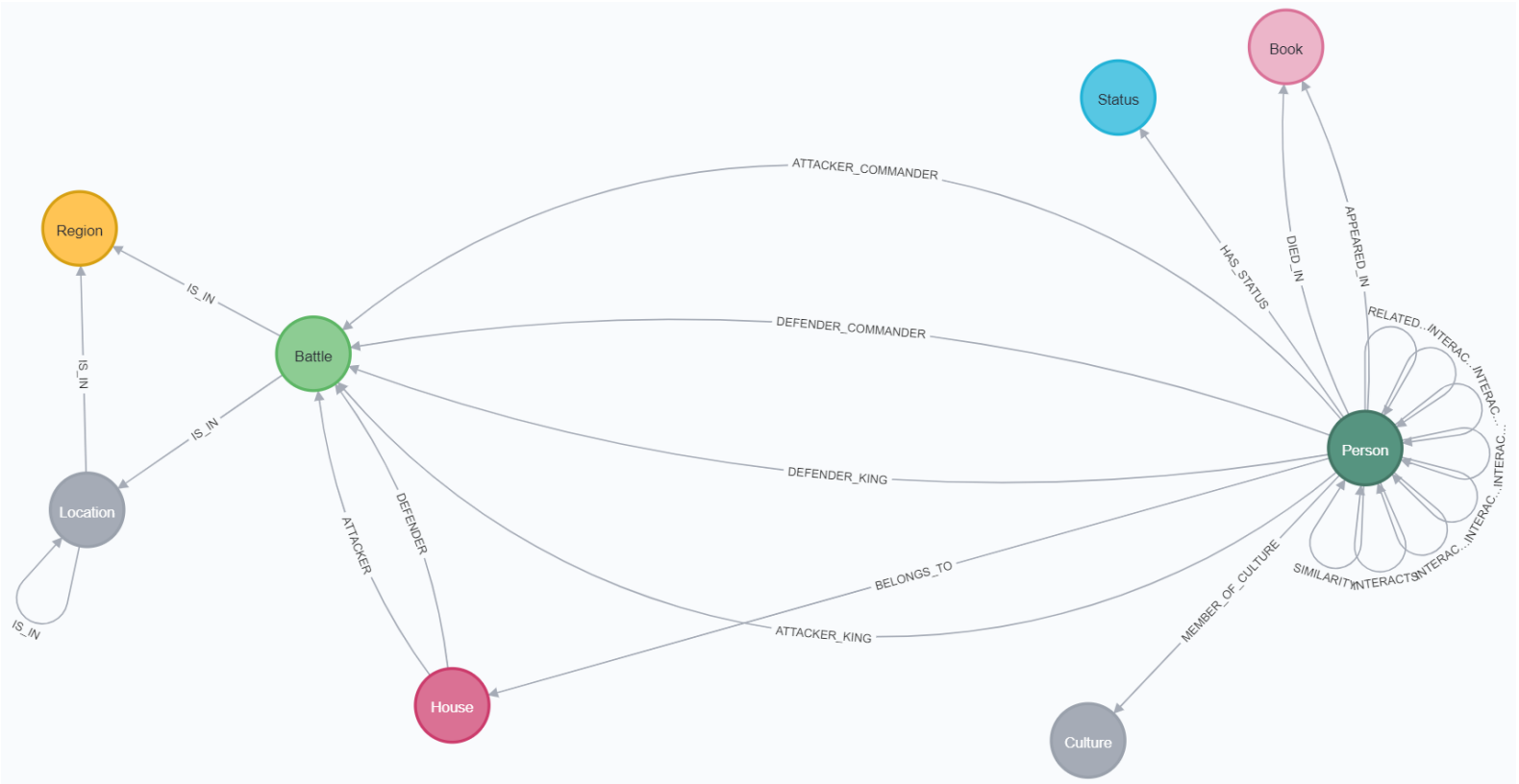

```
LOAD CSV WITH HEADERS FROM 'https://s3.eu-north-1.amazonaws.com/com.neo4j.gds.browser-guide/data/battles.csv' AS row

MERGE (b:Battle {name: row.name})
  ON CREATE SET b.year = toInteger(row.year),
  b.summer = row.summer,
  b.major_death = row.major_death,
  b.major_capture = row.major_capture,
  b.note = row.note,
  b.battle_type = row.battle_type,
  b.attacker_size = toInteger(row.attacker_size),
  b.defender_size = toInteger(row.defender_size);
```

(Refer Full Script and resources on Moodle – battles; character-deaths; character-predictions; asoiaf-book1-edges; asoiaf-book2-edges; asoiaf-book3-edges; asoiaf-book4-edges; asoiaf-book5-edges)

Graph Algorithms and Text - Game of Thrones Data Visualisation



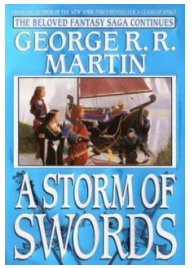
CALL db.schema.visualization()



Remove Dead; King; Knight labels for visualising.
(all appear on Person)

Graph Algorithms and Text - Game of Thrones

Summary Statistics

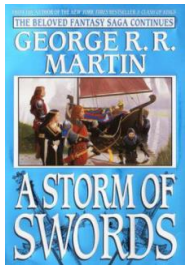



```
MATCH (c:Person)-[:INTERACTS]->()
WITH c, count(*) AS num
RETURN min(num) AS min, max(num) AS max, avg(num) AS avg_interactions, stdev(num) AS stdev
```

	min	max	avg_interactions	stdev
1	1	170	6.782986111111113	14.926129599020737

Graph Algorithms and Text - Game of Thrones

Summary Statistics – by book

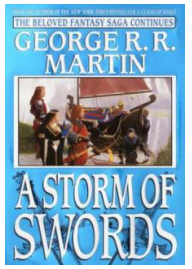



```
MATCH (c:Person)-[r:INTERACTS]->()
WITH r.book AS book, c, count(*) AS num
RETURN book, min(num) AS min, max(num) AS max, avg(num) AS avg_interactions, stdev(num) AS stdev
ORDER BY book
```

book	min	max	avg_interactions	stdev
1	1	51	4.920863309352513	7.096707491528077
2	1	37	4.015544041450774	5.360422756122161
3	1	36	4.299145299145296	5.3596248623847575
4	1	52	3.5894736842105286	5.477215406610821
5	1	45	3.29004329004329	4.758318619541398

Graph Algorithms and Text - Game of Thrones

Estimate memory usage – why?

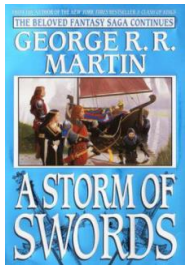


Once you have data and know something about its shape, you need to estimate the memory usage of your graph and algorithm(s), and to configure your Neo4j Server with a much larger heap size than for a transactional deployment. Why?

The graph algorithms run on an in-memory, heap-allocated projection of the Neo4j graph, which resides outside the main database. This means that before you execute an algorithm, you must create (explicitly or implicitly) a projection of your graph in memory. However, creating graphs and running algorithms on them can have a significant memory footprint. Therefore, a good habit is always to estimate the amount of RAM you need and configure a large heap size before running a heavy memory workload.

Graph Algorithms and Text - Game of Thrones

Memory estimation: graphs



To estimate the required memory for a subset of your graph, for example the Person nodes and INTERACTS relationships, call the following procedure.

```
CALL gds.graph.create.estimate('Person', 'INTERACTS') YIELD nodeCount, relationshipCount, requiredMemory
```

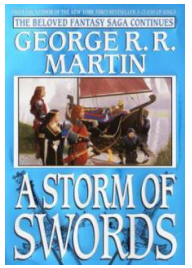
nodeCount	relationshipCount	requiredMemory
2166	3907	"329 KiB"

The result shows estimate is small so you can create the projected graph got-interactions.

```
CALL gds.graph.create('got-interactions', 'Person', 'INTERACTS')
```

Graph Algorithms and Text - Game of Thrones

Estimate memory usage: algorithms



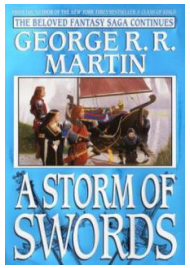
Note that this is for the algorithm only as the graph is already in memory.

```
CALL gds.pageRank.stream.estimate('got-interactions') YIELD requiredMemory
```

requiredMemory
"82 KiB"

Graph Algorithms and Text - Game of Thrones

Estimate memory usage: details



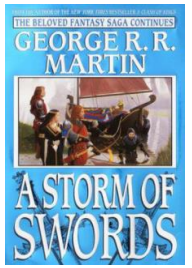
requiredMemory	treeView
"82 KiB"	"Memory Estimation: 82 KiB |-- algorithm: 82 KiB |-- this.instance: 88 Bytes |-- ComputeSteps: 82 KiB |-- this.instance: 32 Bytes |-- scores[] wrapper: 32 Bytes |-- starts[]: 48 Bytes |-- lengths[]: 48 Bytes |-- list of computeSteps: 32 Bytes |-- ComputeStep: 82 KiB |-- this.instance: 128 Bytes |-- nextScores[] wrapper: 32 Bytes |-- inner nextScores[][]: 10496 Bytes |-- pageRank[]: 5224 Bytes |-- deltas[]: 5224 Bytes "

If you want to look at the full details of the memory estimation, remove the YIELD clause. The procedure returns a tree view and a map view of all the "components" with their memory estimates.

```
CALL gds.pageRank.stream.estimate('got-interactions')
```

Graph Algorithms and Text - Game of Thrones

Estimate memory usage: details



requiredMemory	treeView
"410 KiB"	"Memory Estimation: 410 KiB |-- graph: 329 KiB |-- this.instance: 80 Bytes |-- nodeIdMap: 49 KiB |-- this.instance: 40 Bytes |-- Neo4j identifiers: 16 KiB |-- Mapping from Neo4j identifiers to internal identifiers: 32 KiB |-- Node Label BitSets: 312 Bytes |-- adjacency list for 'RelationshipType{name='INTERACTS'}': 256 KiB |-- this.instance: 16 Bytes |-- pages: 256 KiB |-- adjacency offsets for 'RelationshipType{name='INTERACTS'}': 24 KiB |-- this.instance: 32 Bytes |-- pages wrapper: 32 Bytes |-- page[]: 24 KiB |-- algorithm: 81 KiB |-- this.instance: 88 Bytes |-- ComputeSteps: 81 KiB |-- this.instance: 32 Bytes |-- scores[] wrapper: 32 Bytes |-- starts[]: 40 Bytes |-- lengths[]: 40 Bytes |-- list of computeSteps: 32 Bytes |-- ComputeStep: 80 KiB |-- this.instance: 128 Bytes |-- nextScores[] wrapper: 32

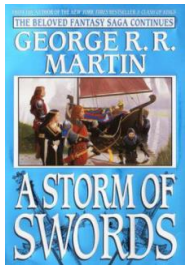
You can also estimate the memory usage for graph creation and algorithm execution at the same time by using the so-called *implicit graph creation*. This way, the configuration for the graph creation is in-lined within the algorithm procedure call.

The result shows an increased memory estimate, explained by the memory consumed by the graph creation.

```
CALL gds.pageRank.stream.estimate({nodeProjection: 'Person',
relationshipProjection: 'INTERACTS'})
```

Graph Algorithms and Text - Game of Thrones

Estimate memory usage: details



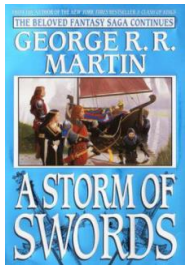
Now, you can filter the result to the top level components: graph and algorithm.

```
CALL gds.pageRank.stream.estimate({
  nodeProjection: 'Person',
  relationshipProjection: 'INTERACTS'
}) YIELD mapView
UNWIND [ x IN mapView.components | [x.name,
x.memoryUsage] ] AS component
RETURN component[0] AS name, component[1] AS size
```

name	size
"graph"	"329 KiB"
"algorithm"	"81 KiB"

Graph Algorithms and Text - Game of Thrones

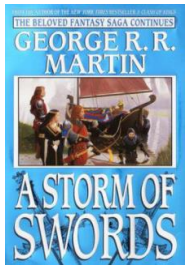
Memory estimation: clean-up



CALL gds.graph.drop('got-interactions');

graphName	database	memoryUsage	sizeInBytes
"got-interactions"	"neo4j"	"853 KiB"	874208

Graph Algorithms and Text - Game of Thrones



Graph creation

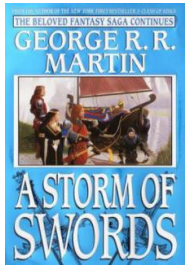
The first stage of execution in GDS is always graph creation, but what does this mean?
To enable fast caching of the graph topology, containing only the relevant nodes, relationships, and weights, the GDS library operates on in-memory graphs that are created as projections of the Neo4j stored graph.
These projections may change the nature of the graph elements by any of the following:

- Sub-graphing
- Renaming relationship types or node labels
- Merging several relationship types or node labels
- Altering relationship direction
- Aggregating parallel relationships and their properties
- Deriving relationships from larger patterns

There are two ways of creating graphs – *explicit* and *implicit*.

Graph Algorithms and Text - Game of Thrones

Graph catalog



The typical workflow is to create the projected graph *explicitly* by giving it a name and storing it in the *graph catalog*. This allows you to operate on the graph multiple times. In *Memory estimation, we* calculated the memory needed for creating a small graph of interactions, called got-interactions. You can create it again. As each INTERACTS relationship is symmetric, you can even ignore its direction by creating your graph with an UNDIRECTED orientation.

```
CALL gds.graph.create('got-interactions', 'Person', {
  INTERACTS: {
    orientation: 'UNDIRECTED'
  }
})
```

Graph Algorithms and Text - Game of Thrones

Graph catalog: standard projection & Cypher projection

The GDS library supports two approaches for loading projected graphs - **standard creation** (gds.graph.create()) and **Cypher projection** (gds.graph.create.cypher()).

In the **standard creation** approach, which you used to create your graph, you specify node labels and relationship types and project them onto the in-memory graph as labels and relationship types with new names. You can further specify properties for each node label and relationship type. For some use cases, this approach might be sufficient. However, it is not possible to take only some nodes with a given label or only some relationships of a given type. One way to work around it is by adding additional labels that define the desired subset of nodes that you want to project.

Graph Algorithms and Text - Game of Thrones

Graph catalog: standard projection & Cypher projection

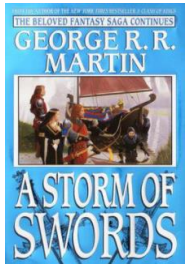
In the **Cypher projection** approach, you use Cypher queries to project nodes and relationships onto the in-memory graph. Instead of specifying labels and relationship types, you define node-statements and relationship-statements. In this way, you can leverage the expressivity of the Cypher language and describe your graph in a more sophisticated way.

It is important to note that the standard creation is orders of magnitude faster than the Cypher projection. When designing a use case with Cypher projection at a production scale, make sure to measure the performance in advance.

We will explore the Cypher projection and load the same graph with a new name, for example, got-interactions-cypher.

Graph Algorithms and Text - Game of Thrones

Graph catalog: Cypher projection




```
CALL gds.graph.create.cypher(
  'got-interactions-cypher',
  'MATCH (n:Person) RETURN id(n) AS id',
  'MATCH (s:Person)-[i:INTERACTS]->(t:Person) RETURN id(s) AS
source, id(t) AS target, i.weight AS weight'
)
```

nodeQuery	relationshipQuery	graphName	nodeCount	relationshipCount	createMillis
"MATCH (n:Person) RETURN id(n) AS id"	"MATCH (s:Person)-[i:INTERACTS]->(t:Person) RETURN id(s) AS source, id(t) AS target, i.weight AS weight"	"got-interactions-cypher"	2166	3907	213

Graph Algorithms and Text - Game of Thrones

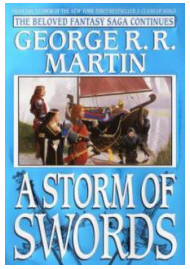
Graph catalog: Cypher projection of virtual relationships

Another interesting feature of the Cypher graph projection is that it allows you to represent complex patterns by computing relationships that do not exist in the Neo4j stored graph. This is especially useful when the algorithm you want to run supports only mono-partite graphs.

For example, you can use the following query to create a graph with Person nodes connected with an (untyped) relationship if they belong to the same house. The projected relationship does not exist in the stored graph.

```
CALL gds.graph.create.cypher(
  'same-house-graph',
  'MATCH (n:Person) RETURN id(n) AS id',
  'MATCH (p1:Person)-[:BELONGS_TO]-(:House)-[:BELONGS_TO]-
(p2:Person) RETURN id(p1) AS source, id(p2) AS target'
)
```

Graph Algorithms and Text - Game of Thrones



Graph catalog: listing

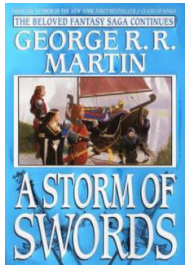
CALL gds.graph.list('got-interactions-cypher')

degreeDistribution	graphName	database	memoryUsage	sizeInBytes
{ "p99": 26, "min": 0, "max": 170, "mean": 1.8037857802400739, "p90": 4, "p50": 0, "p999": 109, "p95": 7, "p75": 1 }	"got-interactions-cypher"	"neo4j"	"1635 KiB"	1674600

CALL gds.graph.list()

Graph Algorithms and Text - Game of Thrones

Graph catalog: existence

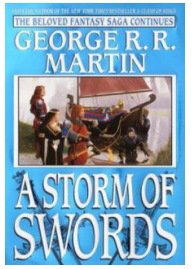


CALL gds.graph.exists('got-interactions')

graphName	exists
"got-interactions"	true

Graph Algorithms and Text - Game of Thrones

Graph catalog: removal

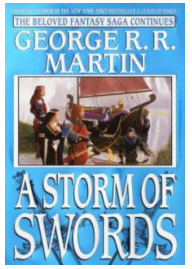


TIP: It is a good practice to remove the unused graphs to free up memory space.

```
CALL gds.graph.drop('got-interactions-cypher');
```

Graph Algorithms and Text - Game of Thrones

Algorithm syntax: explicit graphs



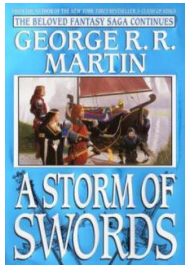
Running algorithms on explicitly created graphs allows you to operate on a graph multiple times. To do this, refer to the graph by its name, as it is stored in the graph catalog.

```
CALL gds.<algo-name>.<mode>(
  graphName: String,
  configuration: Map
)
```

- <algo-name> is the algorithm name.
- <mode> is the algorithm execution mode. The supported modes are:
  - write: writes results to the Neo4j database and returns a summary of the results.
  - stats: same as write but does not write to the Neo4j database.
  - stream: streams results back to the user.
- The graphName parameter value is the name of the graph from the graph catalog.
- The configuration parameter value is the algorithm-specific configuration.

Graph Algorithms and Text - Game of Thrones

Algorithm syntax: implicit graphs



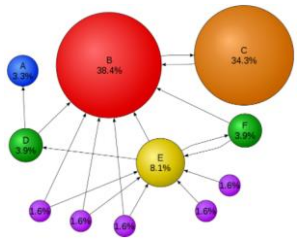
The implicit variant does not access the graph catalog. If you want to run an algorithm on such a graph, you configure the graph creation within the algorithm configuration map.

After the algorithm execution finishes, the graph is released from the memory.

```
CALL gds.<algo-name>.<mode>(
  configuration: Map
)
```

Graph Algorithms and Text - Game of Thrones

PageRank



Page Rank is an algorithm that measures the transitive influence and connectivity of nodes to find the most **influential** nodes in a graph.

It computes an influence value for each node, called a *score*. As a result, the score of a node is a certain weighted average of the scores of its direct neighbours.

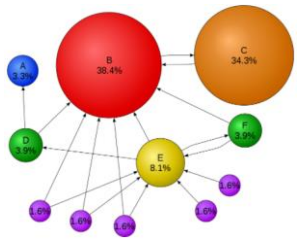
How Page Rank works

PageRank is an *iterative* algorithm. In each iteration, every node propagates its score evenly divided to its neighbours.

The algorithm runs for a configurable maximum number of iterations (default is 20), or until the node scores converge. That is, when the maximum change in node score between two sequential iterations is smaller than the configured tolerance value.

Graph Algorithms and Text - Game of Thrones

PageRank: stream mode



To find out who is influential in the graph we run Page Rank.

```
CALL gds.graph.create('got-interactions', 'Person', {
  INTERACTS: {
    orientation: 'UNDIRECTED'
  }
})
```

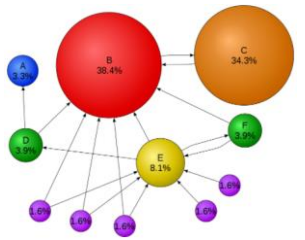
name	score
"Jon Snow"	17.213973485957826
"Tyrion Lannister"	16.862492974847555
"Cersei Lannister"	13.730251107551158
"Jaime Lannister"	13.522420479077846
"Stannis Baratheon"	12.067920422647147
"Daenerys Targaryen"	11.7902354397811
"Arya Stark"	11.221932005137205
"Robb Stark"	10.819954913528635
"Eddard Stark"	10.24471561685204
"Catelyn Stark"	9.997367328964176

Run a basic Page Rank call in stream mode.

```
CALL gds.pageRank.stream('got-interactions') YIELD nodeId,
score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC LIMIT 10
```

Graph Algorithms and Text - Game of Thrones

PageRank: stream mode

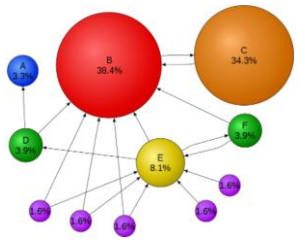


Compare the Page Rank of each Person node with the number of interactions for that node.

name	pageRank	interactions
"Jon Snow"	17.213973485957826	190
"Tyrion Lannister"	16.862492974847555	217
"Cersei Lannister"	13.730251107551158	199
"Jaime Lannister"	13.522420479077846	169
"Stannis Baratheon"	12.067920422647147	147
"Daenerys Targaryen"	11.7902354397811	120
"Arya Stark"	11.221932005137205	132
"Robb Stark"	10.819954913528635	136
"Eddard Stark"	10.24471561685204	126

```
CALL gds.pageRank.stream('got-interactions')
YIELD nodeId, score AS pageRank
WITH gds.util.asNode(nodeId) AS n,
pageRank
MATCH (n)-[i:INTERACTS]-()
RETURN n.name AS name, pageRank,
count(i) AS interactions
ORDER BY pageRank DESC LIMIT 10
```

Graph Algorithms and Text - Game of Thrones



PageRank: write mode

Now that you have the results from your Page Rank query, you write them back to Neo4j and use them for further queries.

You specify the name of the property to which the algorithm will write using the `writeProperty` key in the config map passed to the procedure.

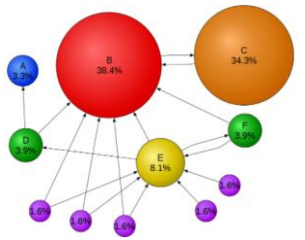
Note that the writing is done in Neo4j, not in the graph `got-interactions`.

```
CALL gds.pageRank.write('got-interactions', {writeProperty: 'pageRank'})
```

writeMillis	nodePropertiesWritten	ranIterations
26	2166	20

Graph Algorithms and Text - Game of Thrones

PageRank: rank per book



Along with the generic INTERACTS relationships, you also have INTERACTS_1, INTERACTS_2, etc., for the different books. We will load a graph for the interactions in book 1 and compute and write the Page Rank scores. Explicit (below middle). Implicit (below right).

```
CALL gds.graph.create(
  'got-interactions-1',
  'Person',
  {
    INTERACTS_1: {
      orientation: 'UNDIRECTED'
    }
  }
);
```

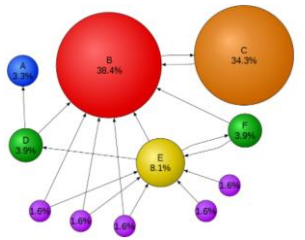
```
CALL gds.pageRank.write(
  'got-interactions-1',
  {
    writeProperty: 'pageRank-1'
  }
)
```

writeMillis	nodePropertiesWritten	ranIterations
44	2166	20

```
CALL gds.pageRank.write({
  nodeProjection: 'Person',
  relationshipProjection: {
    INTERACTS_1: {
      orientation: 'UNDIRECTED'
    }
  },
  writeProperty: 'pageRank-1'
})
```

Graph Algorithms and Text - Game of Thrones

PageRank: Bonus Task



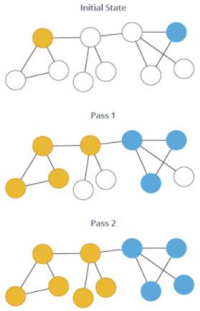
Use a Cypher projection to create a graph of Houses that fought in the same Battles and run Page Rank.

name	score
"Lannister"	2.7469330501742664
"Stark"	2.2027359238156348
"Baratheon"	2.167207883228548
"Greyjoy"	1.5606711879372595
"Tully"	1.2300945719733136
"Frey"	1.1581001617771105
"Bolton"	1.042193059140118
"Karstark"	0.861855861119693
"Glover"	0.861855861119693
"Brave Companions"	0.590372420634958
"Free folk"	0.5776784927991687

```
CALL gds.graph.create.cypher(
  'house-battles',
  'MATCH (h:House) RETURN id(h) AS id',
  'MATCH (h1:House)-->(b:Battle)<--(h2:House) RETURN id(h1)
AS source, id(h2) AS target, count(b) AS weight'
)

CALL gds.pageRank.stream(
  'house-battles',
  {
    relationshipWeightProperty: 'weight'
  }
)
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC
```

Graph Algorithms and Text - Game of Thrones

Label Propagation



Label Propagation (LPA) is a fast algorithm for finding communities in a graph. It propagates labels throughout the graph and forms communities of nodes based on their influence.

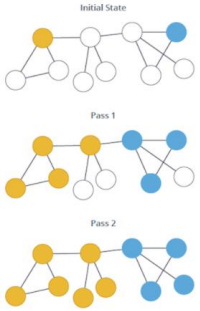
How Label Propagation works

LPA is an *iterative* algorithm. First, it assigns a unique community label to each node. In each iteration, the algorithm changes this label to the most common one among its neighbours. Densely connected nodes quickly broadcast their labels across the graph. At the end of the propagation, only a few labels remain.

Nodes that have the same community label at convergence are considered from the same community. The algorithm runs for a configurable maximum number of iterations, or until it converges.

Graph Algorithms and Text - Game of Thrones

Label Propagation: example

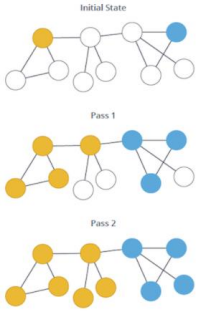


We will run Label Propagation to find the five largest communities of people interacting with each other. For flexibility, in this example, we will create the graph directly in the algorithm call. The weight property on the relationship represents the number of interactions between two people. In LPA, the weight is used to determine the influence of neighbouring nodes when voting on community assignment.

```
CALL gds.graph.create(
  'got-interactions-weighted',
  'Person',
  {
    INTERACTS: {
      orientation: 'UNDIRECTED',
      properties: 'weight'
    }
  }
)
```

Graph Algorithms and Text - Game of Thrones

Label Propagation: example



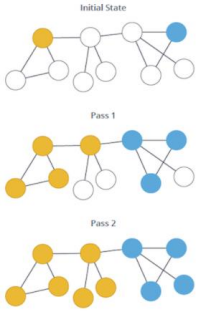
We will run LPA with just one iteration. You can see (below) that the nodes are assigned to initial communities. However, the algorithm needs multiple iterations to achieve a stable result.

```
CALL gds.labelPropagation.stream(
  'got-interactions-weighted',
  {
    relationshipWeightProperty: 'weight',
    maxIterations: 1
  }
) YIELD nodeId, communityId
RETURN communityId, count(nodeId) AS size
ORDER BY size DESC
LIMIT 5
```

communityId	size
304	243
155	90
863	89
332	67
115	53

Graph Algorithms and Text - Game of Thrones

Label Propagation: example



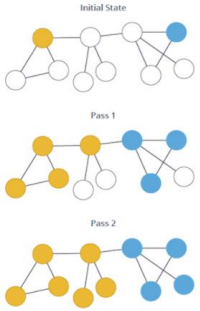

```
CALL gds.labelPropagation.stream(
  'got-interactions-weighted',
  {
    relationshipWeightProperty: 'weight',
    maxIterations: 2
  }
) YIELD nodeId, communityId
RETURN communityId, count(nodeId) AS size
ORDER BY size DESC
LIMIT 5
```

communityId	size
304	331
863	110
155	90
332	79
115	59

Usually, label propagation requires more than a few iterations to converge on a stable result. The number of the required iterations depends on the graph structure — you should experiment.

Graph Algorithms and Text - Game of Thrones

Label Propagation: seeding

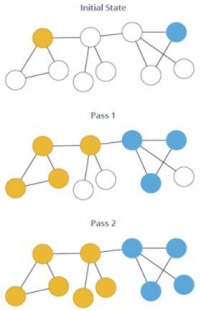


Label Propagation can be seeded with an initial community label from a pre-existing node property. This allows you to compute communities incrementally. We will write the results after the first iteration back to the source graph, under the write property name community.

```
CALL gds.labelPropagation.write(
  'got-interactions-weighted',
  {
    relationshipWeightProperty: 'weight',
    maxIterations: 1,
    writeProperty: 'community'
  }
)
```

Graph Algorithms and Text - Game of Thrones

Label Propagation: seeding

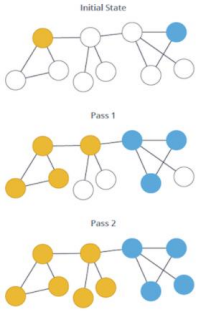


You can use the community property as a seed property for the second iteration. The results should be the same as the previous run with two iterations. Seeding is particularly useful when the source graph grows and you want to compute communities incrementally, without starting again from scratch. Since 'got-interactions-weighted' does not contain the 'community' property, you must create a new graph that does. Right set up the seed.

```
CALL gds.graph.create(
  'got-interactions-seeded',
  {
    Person: {
      properties: 'community'
    }
  },
  {
    INTERACTS: {
      orientation: 'UNDIRECTED',
      properties: 'weight'
    }
  }
)
```

Graph Algorithms and Text - Game of Thrones

Label Propagation: seeding



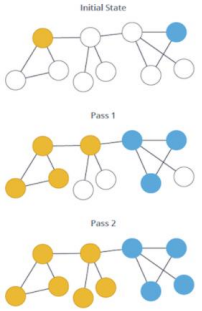
And then, you can use the seed configuration key to specify the property from which you want to seed community IDs.

```
CALL gds.labelPropagation.stream(
  'got-interactions-seeded',
  {
    relationshipWeightProperty: 'weight',
    maxIterations: 1,
    seedProperty: 'community'
  }
) YIELD nodeId, communityId
RETURN communityId, count(nodeId) AS size
ORDER BY size DESC
LIMIT 5
```

communityId	size
304	331
863	110
155	90
332	79
115	59

Graph Algorithms and Text - Game of Thrones

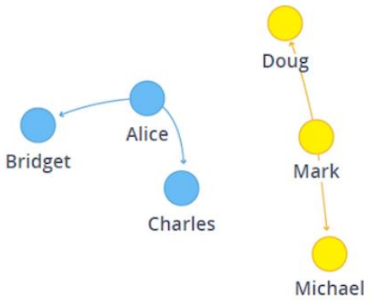
Label Propagation: clean-up




```
CALL gds.graph.drop('got-interactions-weighted');
CALL gds.graph.drop('got-interactions-seeded');
```

Graph Algorithms and Text - Game of Thrones

Weakly Connected Components

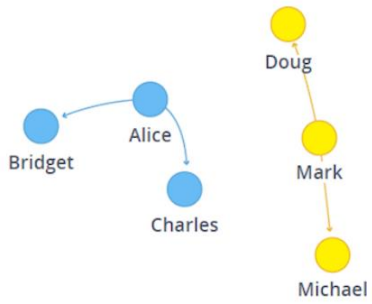


The Weakly Connected Components algorithm (previously known as **Union Find**) finds sets of connected nodes in an *undirected* graph, where each node is reachable from any other node in the same set. It is called *weakly* because it relies on the relationship between two nodes regardless of its direction, wherefore the graph is treated as *undirected*.

This algorithm is useful for identifying disjoint subgraphs, when pre-processing graphs, or for disambiguation purposes.

Graph Algorithms and Text - Game of Thrones

Weakly Connected Components



You can use the got-interactions graph and run the algorithm to compute components.

```
CALL gds.graph.create('got-interactions', 'Person', {
  INTERACTS: {
    orientation: 'UNDIRECTED'
  }
})
```

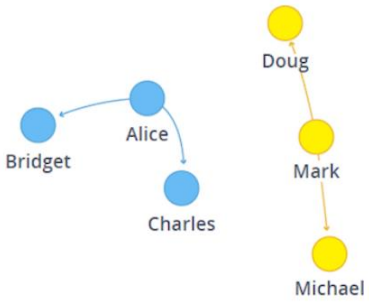
```
CALL gds.wcc.stream('got-interactions')
YIELD nodeId, componentId
RETURN componentId AS component, count(nodeId) AS size
ORDER BY size DESC
```

component	size
0	795
4	1
7	1
8	1
18	1
23	1
30	1
33	1
39	1
41	1
43	1

The result is one large component containing 795 characters and many isolated characters.

Graph Algorithms and Text - Game of Thrones

Weakly Connected Components



We will use a Cypher projection to build a new graph named got-culture-interactions-cypher. It will contain people that belong to the same culture..

```
CALL gds.graph.create.cypher(
  'got-culture-interactions-cypher',
  'MATCH (n:Person) RETURN id(n) AS id',
  'MATCH (p1:Person)-[:MEMBER_OF_CULTURE]->(c:Culture)<-[:MEMBER_OF_CULTURE]-(p2:Person) RETURN id(p1) AS source, id(p2) AS target'
)
```

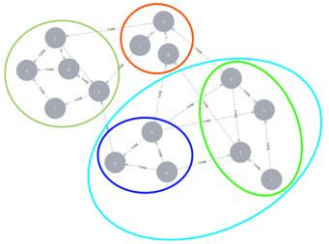
```
CALL gds.wcc.stream('got-culture-interactions-cypher')
YIELD nodeId, componentId
RETURN componentId AS component, count(nodeId) AS size
ORDER BY size DESC
```

```
CALL gds.graph.drop('got-culture-interactions-cypher');
```

component	size
5	133
13	113
38	63
137	43
157	43
116	41
139	26
94	23
114	20
11	19
48	17

Graph Algorithms and Text - Game of Thrones

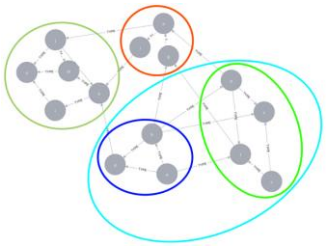
Louvain



The Louvain algorithm, like Label Propagation and Weakly Connected Components, is a community detection algorithm designed to identify clusters of nodes in a graph. It applies heuristic modularity to define the community structure by calculating how densely connected the nodes within a community (module) are, versus in a random graph. Louvain also reveals a hierarchy of communities at different scales, which enables you to zoom in on different levels of granularity and find sub-communities within sub-communities within sub-communities.

Graph Algorithms and Text - Game of Thrones

Louvain



How Louvain works

Louvain is a *greedy*, *hierarchical clustering* algorithm. It repeats the following two steps until it finds a global optimum:

1. Assign the nodes to communities, favouring local optimizations of modularity.
2. Aggregate the nodes from the same community to form a single node, which inherits all connected relationships.

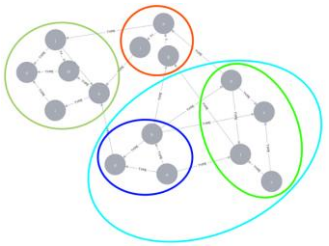
These two steps are repeated until no further modularity-increasing reassignments of communities are possible. Because ties are broken arbitrarily, you can get different results between different runs of the Louvain algorithm.

What to consider

Louvain is significantly slower than Label Propagation and Weakly Connected Components, and the results can be hard to interpret.

Graph Algorithms and Text - Game of Thrones

Louvain



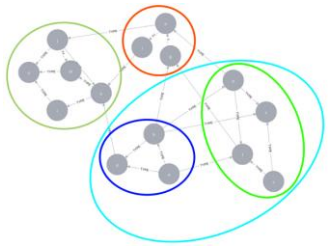
```
CALL gds.graph.create('got-interactions', 'Person', {
  INTERACTS: {
    orientation: 'UNDIRECTED'
  }
})

CALL gds.louvain.stream('got-interactions')
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name AS person,
communityId
ORDER BY communityId DESC
```

person	communityId
"Aemon Targaryen"	2092
"Durran the Devout"	2091
"Baelor 'Breakspear' Targaryen"	2090
"Arlan V Durrandon"	2089
"Aenys Targaryen"	2088
"Jaehaerys Targaryen"	2087
"Halleck Hoare"	2085
"Qhorwyn Hoare"	2084
"Baelon Targaryen"	2083
"Jeyne Marbrand"	2082
"Charlton"	2081

The query returns the name of each person and the id of the community to which it belongs.

Graph Algorithms and Text - Game of Thrones

Louvain



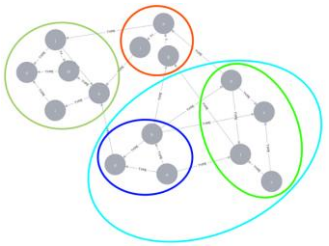
If you want to investigate how many communities are available, and the number of members of each community, you can change the RETURN statement. The result is 1382 communities, 11 of which with more than one member.

```
CALL gds.louvain.stream('got-interactions')
YIELD nodeId, communityId
RETURN communityId, COUNT(DISTINCT nodeId) AS members
ORDER BY members DESC
```

communityId	members
1202	162
205	155
77	118
233	112
706	67
20	60
46	45
1966	32
636	26
1857	11
1890	7

Graph Algorithms and Text - Game of Thrones

Louvain - weighting

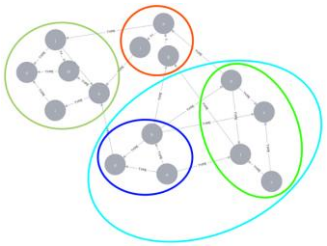


We will run the Louvain algorithm on a weighted graph. This way, it considers the relationship weights when calculating the modularity.

First, create a graph with the weight relationship property. Otherwise, the number specified in defaultValue will be used as a fallback.

Graph Algorithms and Text - Game of Thrones

Louvain - weighting



```
CALL gds.graph.create(
  'got-weighted-interactions',
  'Person',
  {
    INTERACTS: {
      orientation: 'UNDIRECTED',
      aggregation: 'NONE',
      properties: {
        weight: {
          property: 'weight',
          aggregation: 'NONE',
          defaultValue: 0.0
        }
      }
    }
  }
)
```

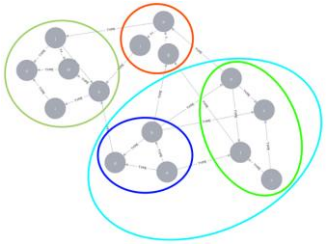
Using the weight property on the INTERACTS relationship.:

```
CALL gds.louvain.stream(
  'got-weighted-interactions',
  {
    relationshipWeightProperty: 'weight'
  }
)
YIELD nodeId, communityId
RETURN communityId, COUNT(DISTINCT nodeId) AS members
ORDER BY members DESC
```

The result is 1384 communities, 13 of which with more than one member.

Graph Algorithms and Text - Game of Thrones

Louvain intermediate communities



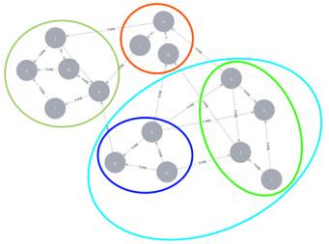
We will try to identify communities at multiple levels in the graph: first small communities, and then combine them in large ones.

To retrieve the intermediate communities,
set **includeIntermediateCommunities** to true.

```
CALL gds.louvain.stream(
  'got-interactions',
  {
    includeIntermediateCommunities: true
  }
)
YIELD nodeId, communityId, intermediateCommunityIds
RETURN communityId, COUNT(DISTINCT nodeId) AS members,
intermediateCommunityIds
```

Graph Algorithms and Text - Game of Thrones

Louvain intermediate communities

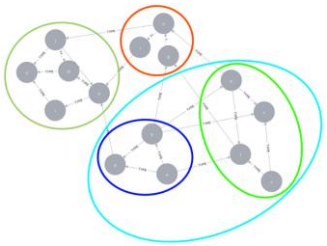


communityId	members	intermediateCommunityIds
205	36	[73, 205, 205]
1202	17	[284, 532, 1202]
205	30	[1118, 205, 205]
706	24	[706, 706, 706]
4	1	[4, 4, 4]
1202	20	[10, 1202, 1202]
7	1	[7, 7, 7]
8	1	[8, 8, 8]

Sample of the output

Graph Algorithms and Text - Game of Thrones

Louvain intermediate communities



```
CALL gds.louvain.stream(
  'got-interactions',
  {
    includeIntermediateCommunities: true
  }
)
YIELD nodeId, intermediateCommunityIds
RETURN count(distinct intermediateCommunityIds[0]),
count(distinct intermediateCommunityIds[1])
```

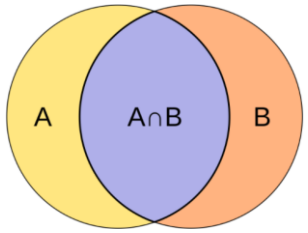
You can extract membership in different levels of communities and see how the composition changes. includeIntermediateCommunities: false is the default value, in which case,the intermediateCommunityIds field of the result is null. Clean-up (drop projected graph) when analysis completed.

count(distinct intermediateCommunityIds[0])	count(distinct intermediateCommunityIds[1])
1460	1383

```
CALL gds.graph.drop('got-weighted-interactions');
```

Graph Algorithms and Text - Game of Thrones

Node Similarity



The Node Similarity algorithm compares pairs of nodes in a graph based on their connections to other nodes. Two nodes are considered similar if they share many of the same neighbours.

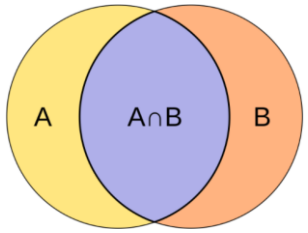
The algorithm uses the so-called *Jaccard Similarity Score* to obtain a similarity measure between two sets. More precisely, the similarity between two nodes A and B is given by the following formula:

$$Similarity(A, B) = \frac{[\#nodes\ neighbouring\ A\ and\ B]}{\#nodes\ neighbouring\ A\ or\ B\ (or\ both)}$$

That is, nodes A and B are similar if most nodes that are neighbours to either node are also neighbours to both.

Graph Algorithms and Text - Game of Thrones

Node Similarity

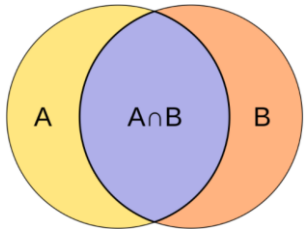


How it works

The input of this algorithm is a bi-partite, connected graph containing two disjoint node sets. Each relationship starts from a node in the first node set and ends at a node in the second node set. The Node Similarity algorithm compares all nodes from the first node set with each other based on their relationships to nodes in the second set. The complexity of this comparison grows quadratically with the number of nodes to compare. The algorithm reduces the complexity by ignoring disconnected nodes.

Graph Algorithms and Text - Game of Thrones

Node Similarity – example graph



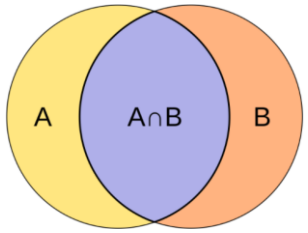
Before you run the Node Similarity algorithm, you have to create a projected graph that consists of GOT characters and the various entities to which they relate. The task will be to find similar characters by comparing the books they appear or die in, and the houses and cultures to which they belong. It is a bipartite graph between Person on one side and Book, House, and Culture on the other side.

This graph creation uses projection with multiple node labels. You load all types of relationships with *.

```
CALL gds.graph.create('got-character-related-entities', ['Person', 'Book', 'House', 'Culture'], '*')
```

Graph Algorithms and Text - Game of Thrones

Node Similarity – simple run

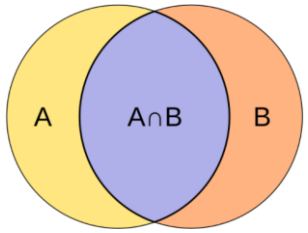


Now, you can run Node Similarity with the default settings and extract the top 10 most similar pairs of characters. The algorithm computes similarities only for Person nodes as they are the only nodes with outgoing edges. To get more interesting results, you can limit the result by using the property degreeCutoff, to get only characters with at least 20 related entities.

```
CALL gds.nodeSimilarity.stream(
  'got-character-related-entities',
  {
    degreeCutoff: 20
  }
)
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS character1,
gds.util.asNode(node2).name AS character2, similarity
ORDER BY similarity DESC
LIMIT 10
```

Graph Algorithms and Text - Game of Thrones

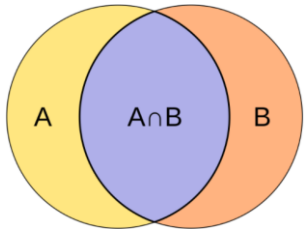
Node Similarity – simple run



character1	character2	similarity
"Ilyn Payne"	"Gregor Clegane"	0.5
"Gregor Clegane"	"Ilyn Payne"	0.5
"Loras Tyrell"	"Mace Tyrell"	0.4666666666666667
"Mace Tyrell"	"Loras Tyrell"	0.4666666666666667
"Loras Tyrell"	"Margaery Tyrell"	0.45161290322580644
"Margaery Tyrell"	"Loras Tyrell"	0.45161290322580644
"Brynden Tully"	"Edmure Tully"	0.4166666666666667
"Edmure Tully"	"Brynden Tully"	0.4166666666666667
"Amory Lorch"	"Gregor Clegane"	0.41025641025641024
"Gregor Clegane"	"Amory Lorch"	0.41025641025641024

Graph Algorithms and Text - Game of Thrones

Node Similarity – similarity cut-off

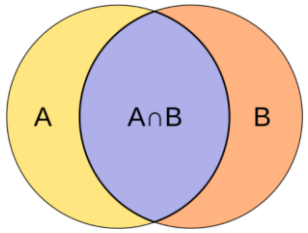


In most real-world graphs, the number of pairs of nodes to compare is huge, and most pairs are not similar. Therefore, it is useful to be able to limit the output. There are several ways to deal with this. One way is to set a threshold for a minimum similarity by specifying the similarityCutoff property.

Note that you no longer need to use the LIMIT clause. By default, the similarityCutoff value is a very small number, effectively filtering out pairs that have zero similarity.

Graph Algorithms and Text - Game of Thrones

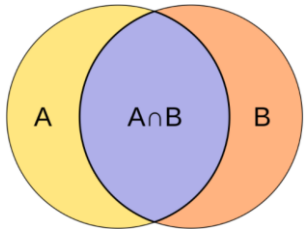
Node Similarity – similarity cut-off




```
CALL gds.nodeSimilarity.stream(
  'got-character-related-entities',
  {
    degreeCutoff: 20,
    similarityCutoff: 0.45
  }
)
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS character1,
gds.util.asNode(node2).name AS character2, similarity
ORDER BY similarity DESC
```

Graph Algorithms and Text - Game of Thrones

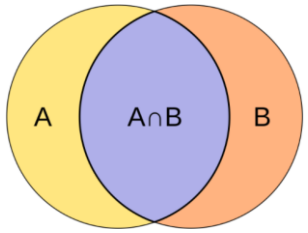
Node Similarity – similarity cut-off



character1	character2	similarity
"Gregor Clegane"	"Ilyn Payne"	0.5
"Ilyn Payne"	"Gregor Clegane"	0.5
"Mace Tyrell"	"Loras Tyrell"	0.4666666666666667
"Loras Tyrell"	"Mace Tyrell"	0.4666666666666667
"Loras Tyrell"	"Margaery Tyrell"	0.45161290322580644
"Margaery Tyrell"	"Loras Tyrell"	0.45161290322580644

Graph Algorithms and Text - Game of Thrones

Node Similarity – topN

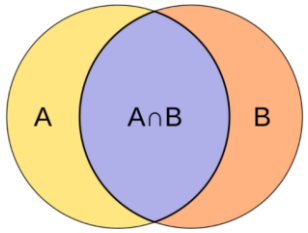


You can also limit the number of similarities returned by using the topN config option. This algorithm specific way of limiting is more memory efficient than constructing the entire stream and using the LIMIT clause afterwards.

```
CALL gds.nodeSimilarity.stream(
  'got-character-related-entities',
  {
    degreeCutoff: 20,
    topN: 10
  }
)
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS character1,
gds.util.asNode(node2).name AS character2, similarity
ORDER BY similarity DESC
```

Graph Algorithms and Text - Game of Thrones

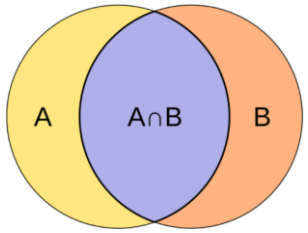
Node Similarity – topN



character1	character2	similarity
"Ilyn Payne"	"Gregor Clegane"	0.5
"Gregor Clegane"	"Ilyn Payne"	0.5
"Loras Tyrell"	"Mace Tyrell"	0.4666666666666667
"Mace Tyrell"	"Loras Tyrell"	0.4666666666666667
"Margaery Tyrell"	"Loras Tyrell"	0.45161290322580644
"Loras Tyrell"	"Margaery Tyrell"	0.45161290322580644
"Edmure Tully"	"Brynden Tully"	0.4166666666666667
"Brynden Tully"	"Edmure Tully"	0.4166666666666667
"Amory Lorch"	"Gregor Clegane"	0.41025641025641024
"Gregor Clegane"	"Amory Lorch"	0.41025641025641024

Graph Algorithms and Text - Game of Thrones

Node Similarity – topK

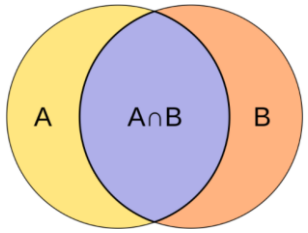


Another way to limit the results is the topK config option. The algorithm output will be the K most similar characters for each character. Let's set this value to 1, and see if Loras Tyrell has only one similar neighbour instead of two.

```
CALL gds.nodeSimilarity.stream(
  'got-character-related-entities',
  {
    degreeCutoff: 20,
    topN: 10,
    topK: 1
  }
)
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS character1,
gds.util.asNode(node2).name AS character2, similarity
ORDER BY similarity DESC
```

Graph Algorithms and Text - Game of Thrones

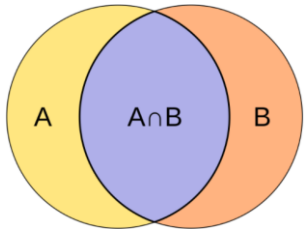
Node Similarity – topK



character1	character2	similarity
"Ilyn Payne"	"Gregor Clegane"	0.5
"Gregor Clegane"	"Ilyn Payne"	0.5
"Loras Tyrell"	"Mace Tyrell"	0.4666666666666667
"Mace Tyrell"	"Loras Tyrell"	0.4666666666666667
"Margaery Tyrell"	"Loras Tyrell"	0.45161290322580644
"Edmure Tully"	"Brynden Tully"	0.4166666666666667
"Brynden Tully"	"Edmure Tully"	0.4166666666666667
"Amory Lorch"	"Gregor Clegane"	0.41025641025641024
"Balon Swann"	"Ilyn Payne"	0.40625
"Bowen Marsh"	"Aemon Targaryen (son of Maekar I)"	0.40540540540540543

Graph Algorithms and Text - Game of Thrones

Node Similarity – topK

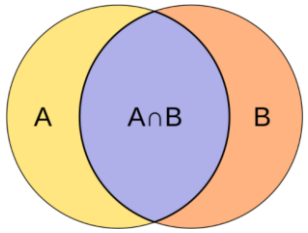


Previous slide. Loras Tyrell still appeared twice as character2.

The algorithm returns only the most similar character to Loras when considering his neighbours. The explanation is that when considering other characters, multiple ones may have Loras as their most similar neighbour.

Graph Algorithms and Text - Game of Thrones

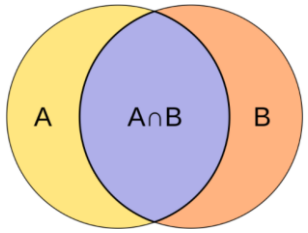
Node Similarity – bottomN and bottomK



Similarly to the topN and topK, bottomN and bottomK config options limit the results but return the least similar pairs.

Graph Algorithms and Text - Game of Thrones

Node Similarity – writing

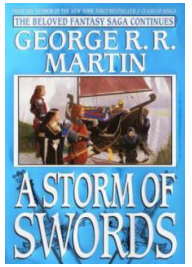


The output of the algorithm can be written as weighted relationships. The weight property is set to the computed node similarity of the relationship it concerns. The config option writeProperty specifies the name of the property. The result is 10 relationships caused by the topN value.

```
CALL gds.nodeSimilarity.write(
  'got-character-related-entities',
  {
    degreeCutoff: 20,
    topN: 10,
    topK: 1,
    writeRelationshipType: 'SIMILARITY',
    writeProperty: 'character_similarity'
  }
)
```

Graph Algorithms and Text - Game of Thrones

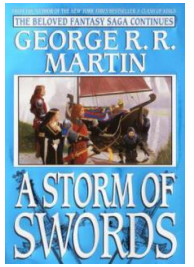
Triangle Count



A triangle in a graph is a set of three nodes all connected to each other. The triangle count of a node is the number of triangles that node belongs to. The Graph Data Science library provides procedures for all standard execution modes in the namespace gds.triangleCount. The algorithm is only defined for undirected graphs, so we make sure to fulfil this requirement in the examples below.

Graph Algorithms and Text - Game of Thrones

Triangle Count - examples

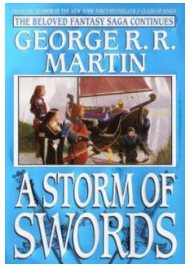


A triangle in a graph is a set of three nodes all connected to each other. The triangle count of a node is the number of triangles that node belongs to. The Graph Data Science library provides procedures for all standard execution modes in the namespace gds.triangleCount. The algorithm is only defined for undirected graphs, so we make sure to fulfil this requirement in the examples below.

```
MATCH (n:Person)-[r:INTERACTS_1]->(m:Person)
WHERE n.name IN ["Robb Stark", "Tyrion Lannister"]
RETURN n, m, r
```

Graph Algorithms and Text - Game of Thrones

Triangle Count - examples

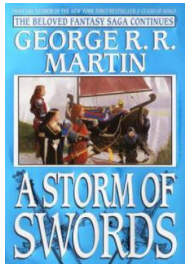



```
CALL gds.graph.create.cypher('small_got',
'MATCH (n:Person) RETURN id(n) AS id',
"MATCH (n:Person)-[r:INTERACTS_1]->(m:Person)
  WHERE n.name IN ['Robb Stark', 'Tyrion Lannister'] RETURN
id(n) AS source, id(m) AS target
  UNION MATCH (n:Person)-[r:INTERACTS_1]->(m:Person)
WHERE n.name IN ['Robb Stark', 'Tyrion Lannister'] RETURN
id(m) AS source, id(n) AS target")
```

nodeQuery	relationshipQuery	graphName	nodeCount	relationshipCount
"MATCH (n:Person) RETURN id(n) AS id"	"MATCH (n:Person)-[r:INTERACTS_1]->(m:Person) WHERE n.name IN ['Robb Stark', 'Tyrion Lannister'] RETURN id(n) AS source, id(m) AS target UNION MATCH (n:Person)-[r:INTERACTS_1]->(m:Person) WHERE n.name IN ['Robb Stark', 'Tyrion Lannister'] RETURN id(m) AS source, id(n) AS target"	"small_got"	2166	36

Graph Algorithms and Text - Game of Thrones

Triangle Count - examples



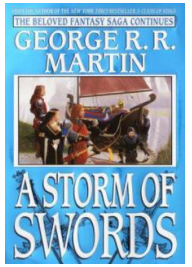

```
CALL gds.triangleCount.stream('small_got')
YIELD nodeId, triangleCount
WITH gds.util.asNode(nodeId).name AS name, triangleCount
WHERE triangleCount > 0
RETURN name, triangleCount
```

name	triangleCount
"Robb Stark"	2
"Tywin Lannister"	1
"Tyrion Lannister"	2
"Yoren"	1

There are exactly two triangles which give Tyrion and Robb triangle counts of two and Tywin and Yoren triangle counts of one.

Graph Algorithms and Text - Game of Thrones

Triangle Count - examples



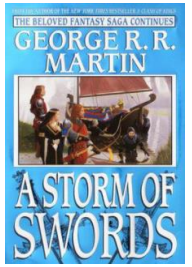
We look to find the people with the highest overall triangle count in book 1.

```
CALL gds.graph.create(
  'got-interactions-1',
  'Person',
  {
    INTERACTS_1: {
      orientation: 'UNDIRECTED'
    }
  }
);
```

```
CALL gds.triangleCount.stream('got-interactions-1')
YIELD nodeId, triangleCount
RETURN gds.util.asNode(nodeId).name AS name, triangleCount
ORDER BY triangleCount DESC
LIMIT 10
```

Graph Algorithms and Text - Game of Thrones

Triangle Count - examples

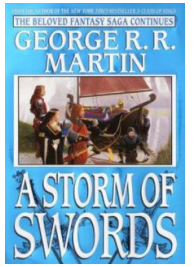


name	triangleCount
"Eddard Stark"	285
"Robert Baratheon"	260
"Sansa Stark"	199
"Tyrion Lannister"	187
"Joffrey Baratheon"	183
"Cersei Lannister"	179
"Catelyn Stark"	175
"Petyr Baelish"	155
"Robb Stark"	154
"Bran Stark"	149

Does perhaps Eddard Stark have an inclination to triangle dramas?

Graph Algorithms and Text - Game of Thrones

Triangle Count – stats mode

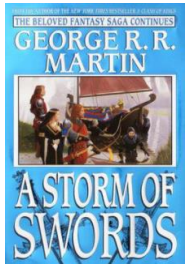


CALL gds.triangleCount.stats('got-interactions-1')
YIELD globalTriangleCount

globalTriangleCount
1480

Graph Algorithms and Text - Game of Thrones

Triangle Count – max degree

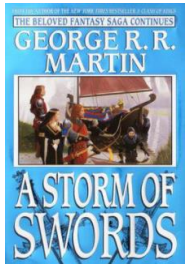


For nodes with a high degree, it is expensive to compute the triangle count. One can exclude certain nodes from the computation by setting the configuration option maxDegree as follows. For each excluded node, the triangle count will be reported as -1. These nodes will also be excluded from the triangle counts of the adjacent nodes.

```
CALL gds.triangleCount.stream('got-interactions-1',
{maxDegree: 10})
YIELD nodeId, triangleCount
WHERE triangleCount <> 0
RETURN gds.util.asNode(nodeId).name AS name, triangleCount
LIMIT 20
```

Graph Algorithms and Text - Game of Thrones

Triangle Count – max degree

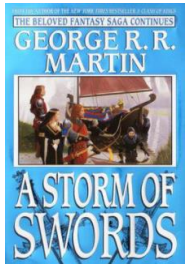


name	triangleCount
"Jaime Lannister"	-1
"Gregor Clegane"	-1
"Roose Bolton"	3
"Robb Stark"	-1
"Brynden Tully"	-1
"Theon Greyjoy"	-1
"Rodrik Cassel"	-1
"Stannis Baratheon"	-1
"Tywin Lannister"	-1
"Loras Tyrell"	-1
"Edmure Tully"	-1

We note that for example Eddard Stark is no longer in the top list of high triangle count characters since his degree 51 exceeds the maxDegree setting. Moreover, the triangle counts for nodes of lower degrees are also affected.

Graph Algorithms and Text - Game of Thrones

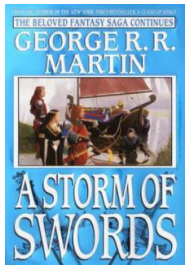
Triangle Count – clean-up




```
CALL gds.graph.drop('got-interactions-1');
```

Graph Algorithms and Text - Game of Thrones

Triangle Count – Local Clustering Coefficient



The local clustering coefficient is a metric quantifying how connected the neighbourhood of a node is. It is the probability that two random neighbours of the node are connected in the graph. This can be obtained from the triangle count and the degree of the node. The Graph Data Science library provides procedures for all standard execution modes in the namespace gds.localClusteringCoefficient. The algorithm is only defined for undirected graphs, so we make sure to fulfil this requirement in the examples below.

Graph Algorithms and Text - Game of Thrones

Triangle Count – Local Clustering Coefficient - stream

```
CALL gds.graph.create(
  'got-interactions-1',
  'Person',
  {
    INTERACTS_1: {
      orientation: 'UNDIRECTED'
    }
  }
);
```

```
CALL gds.localClusteringCoefficient.stream('got-interactions-1')
YIELD nodeId, localClusteringCoefficient
RETURN gds.util.asNode(nodeId).name AS name,
localClusteringCoefficient
ORDER BY localClusteringCoefficient DESC
LIMIT 10
```

Graph Algorithms and Text - Game of Thrones

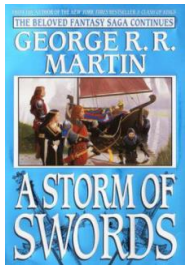
Triangle Count – Local Clustering Coefficient - stream

name	localClusteringCoefficient
"Boros Blount"	1.0
"Chett"	1.0
"Balon Greyjoy"	1.0
"Balon Swann"	1.0
"Bowen Marsh"	1.0
"Albett"	1.0
"Kevan Lannister"	1.0
"High Septon (fat_one)"	1.0
"Addam Marbrand"	1.0
"Chiggen"	1.0

We see here multiple nodes with local clustering coefficient 1.0, however they have in fact only few neighbours and triangles, sometimes a single triangle. In the following example we will identify nodes with high local clustering coefficient but filter out nodes with low triangle count.

Graph Algorithms and Text - Game of Thrones

Triangle Count – triangleCountProperty

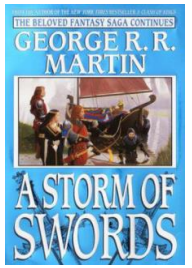


To compute the local clustering coefficient we need to know the number of triangles for each node. The Local Clustering Coefficient is capable of reusing previously computed triangle counts.
First we compute the triangle counts and save them in the in-memory graph as a node property called triangleCount.

```
CALL gds.triangleCount.mutate('got-interactions-1',
{mutateProperty: "triangleCount"})
```

Graph Algorithms and Text - Game of Thrones

Triangle Count – triangleCountProperty

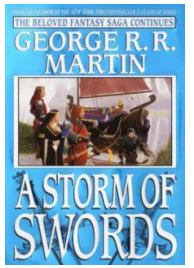


In the following, we look at nodes which have both a high triangle count and a high local clustering coefficient. The persons we see here might be regarded as central in medium to large communities.

```
CALL gds.localClusteringCoefficient.stream('got-interactions-1',
{triangleCountProperty: "triangleCount"})
YIELD nodeId, localClusteringCoefficient AS lcc
WITH gds.util.asNode(nodeId).name AS name , lcc,
gds.util.nodeProperty('got-interactions-1', nodeId,
"triangleCount") AS triangleCount
WHERE triangleCount > 50
RETURN name, lcc, triangleCount
ORDER BY lcc DESC
LIMIT 10
```

Graph Algorithms and Text - Game of Thrones

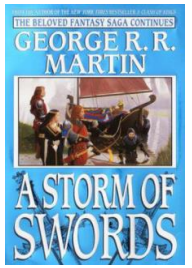
Triangle Count – triangleCountProperty



name	lcc	triangleCount
"Barristan Selmy"	0.7472527472527473	68.0
"Pycelle"	0.7362637362637363	67.0
"Varys"	0.6838235294117647	93.0
"Sandor Clegane"	0.625	75.0
"Renly Baratheon"	0.5947712418300654	91.0
"Jon Arryn"	0.580952380952381	61.0
"Theon Greyjoy"	0.5428571428571428	57.0
"Rodrik Cassel"	0.5032679738562091	77.0
"Petyr Baelish"	0.47692307692307695	155.0
"Joffrey Baratheon"	0.4206896551724138	183.0

Graph Algorithms and Text - Game of Thrones

Local Clustering Coefficient – stats mode



To see if the GoT person graph of book 1 is a small-world network, we can run the following Cypher code.

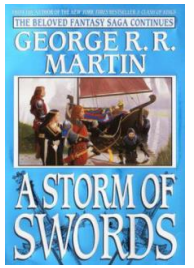
```
CALL gds.localClusteringCoefficient.stats('got-interactions-1')
YIELD averageClusteringCoefficient
```

averageClusteringCoefficient
0.04421340812802044

As we see, the average clustering coefficient of around 0.036 is rather small. In comparison, clustering coefficients of 0.11 have been reported for the world wide web and 0.59 for a network of company directors. The explanation for the lack of small world structure could be that there are many characters in GoT, and it would require even more pages to turn them into a small-world network.

Graph Algorithms and Text - Game of Thrones

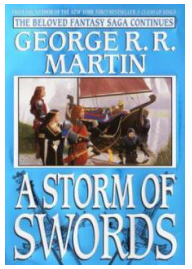
Local Clustering Coefficient – stats mode



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

Graph Algorithms and Text - Game of Thrones

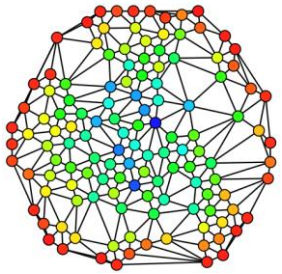
Local Clustering Coefficient – clean-up




```
CALL gds.graph.drop('got-interactions-1');
```

Graph Algorithms and Text - Game of Thrones

Betweenness Centrality

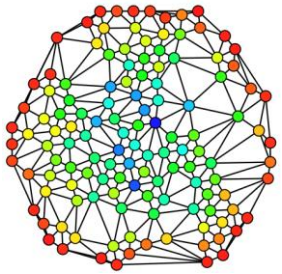


Betweenness Centrality is a way of detecting the amount of influence a node has over the flow of information in a graph. It is often used to find nodes that serve as a bridge from one part of a graph to another.

How Betweenness Centrality works
The algorithm calculates unweighted shortest paths between all pairs of nodes in a graph. Each node receives a score, based on the number of shortest paths that pass through the node. Nodes that more frequently lie on shortest paths between other nodes will have higher betweenness centrality scores.

Graph Algorithms and Text - Game of Thrones

Betweenness Centrality



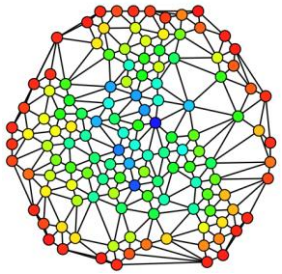
We find out who is influential in the graph by running Betweenness Centrality.

```
CALL gds.graph.create('got-interactions', 'Person', {
  INTERACTS: {
    orientation: 'UNDIRECTED'
  }
})
```

```
CALL gds.betweenness.stream('got-interactions') YIELD nodeId,
score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC LIMIT 10
```

Graph Algorithms and Text - Game of Thrones

Betweenness Centrality



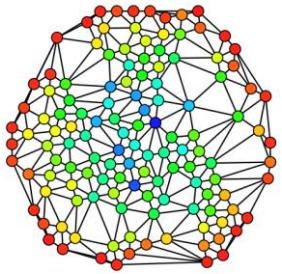
name	score
"Jon Snow"	17.213973485957826
"Tyrion Lannister"	16.862492974847555
"Cersei Lannister"	13.730251107551158
"Jaime Lannister"	13.522420479077846
"Stannis Baratheon"	12.067920422647147
"Daenerys Targaryen"	11.7902354397811
"Arya Stark"	11.221932005137205
"Robb Stark"	10.819954913528635
"Eddard Stark"	10.24471561685204
"Catelyn Stark"	9.997367328964176

The result is similar, but not identical to PageRank (next slide).

In general Betweenness Centrality is a good metric to identify bottlenecks and bridges in a graph while Page Rank is used to understand the influence of a node in a network.

Graph Algorithms and Text - Game of Thrones

Betweenness Centrality

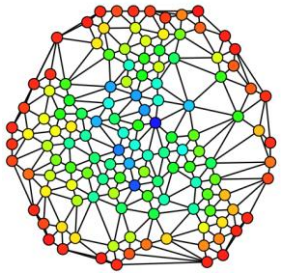


name	score
"Jon Snow"	65746.10756233714
"Tyrion Lannister"	49958.79998213848
"Daenerys Targaryen"	38022.02252425337
"Stannis Baratheon"	37406.523703628
"Theon Greyjoy"	35336.02748203102
"Jaime Lannister"	32173.227032450202
"Robert Baratheon"	30639.619775141666
"Arya Stark"	29160.623805253716
"Cersei Lannister"	28932.56094710855
"Eddard Stark"	26573.000548222735

```
CALL gds.pageRank.stream('got-interactions')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS
name, score
ORDER BY score DESC LIMIT 10
```

Graph Algorithms and Text - Game of Thrones

Betweenness Centrality - sampling



This algorithm is very computationally expensive. To make it possible to run on large graphs we sample. Sampling means we compute the shortest paths for some nodes but not for others. The number of nodes sampled is configured using the samplingSize parameter.

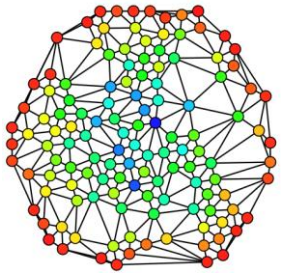
Find out how many nodes are in your graph:

```
CALL gds.graph.list('got-interactions') YIELD nodeCount
```

nodeCount
2166

Graph Algorithms and Text - Game of Thrones

Betweenness Centrality - sampling

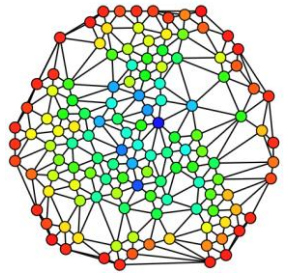


Decide how large of a sample to use. Here we run with half the node count as sampleSize. The appropriate sample size for a use case will depend on the size and shape of the graph, as well as the resources (RAM and CPU) available.

```
CALL gds.betweenness.stream('got-interactions', {samplingSize: 1083})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score
ORDER BY score DESC LIMIT 10
```

Graph Algorithms and Text - Game of Thrones

Betweenness Centrality - sampling



name	score
"Jon Snow"	31072.46508096898
"Tyrion Lannister"	22321.910020282394
"Daenerys Targaryen"	18651.40651376732
"Stannis Baratheon"	18574.08673903996
"Theon Greyjoy"	17555.64208828454
"Jaime Lannister"	15038.634411446557
"Cersei Lannister"	15013.870928607552
"Robert Baratheon"	14759.609134207602
"Eddard Stark"	13901.019009640344
"Arya Stark"	12959.948131578061

Graph Algorithms and Text - Game of Thrones

Betweenness Centrality – stats, write and mutate

In stats mode, Betweenness Centrality will return statistical and measurement values of the centrality score.

The same is returned by the write and mutate modes as well, in addition to writing results back to Neo4j or mutating the in-memory graph, respectively.

centralityDistribution


```
{
  "p99": 7157.062499523163,
  "min": 0.0,
  "max": 65746.49999952316,
  "mean": 351.840951983178,
  "p90": 76.74316358566284,
  "p50": 0.0,
  "p999": 38022.24999952316,
  "p95": 793.0039057731628,
  "p75": 0.0
}
```


```
CALL gds.betweenness.stats('got-interactions')
YIELD centralityDistribution
```