Graph algorithms in Neo4j

Examples and data from OReilly's Graph Algorithms Book

(Posted in ICON)

AND https://neo4j.com/docs/graph-data-science/

What is the Graph Data Science Library?

- Open Source Neo4j Add-On for graph analytics
- Provides a set of high performance graph algorithms
  - Community Detection / Clustering (e.g. Label Propagation)
  - Similarity Calculation (e.g. NodeSimilarity)
  - Centrality Algorithms (e.g. PageRank)
  - PathFinding (e.g. Dijkstra)
  - Link Prediction (e.g. AdamicAdar)
  - and more

Local Patterns to Global Computation

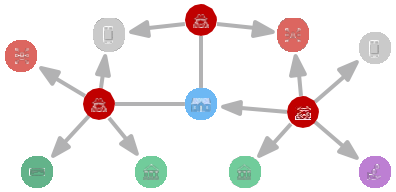
Query (e.g. Cypher/SQL)

Real-time, local decisioning and pattern matching

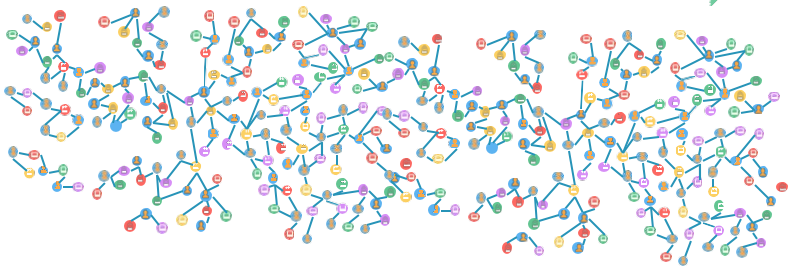
Graph Algorithms Libraries

Global analysis and iterations

Local Patterns → **Global Computation**

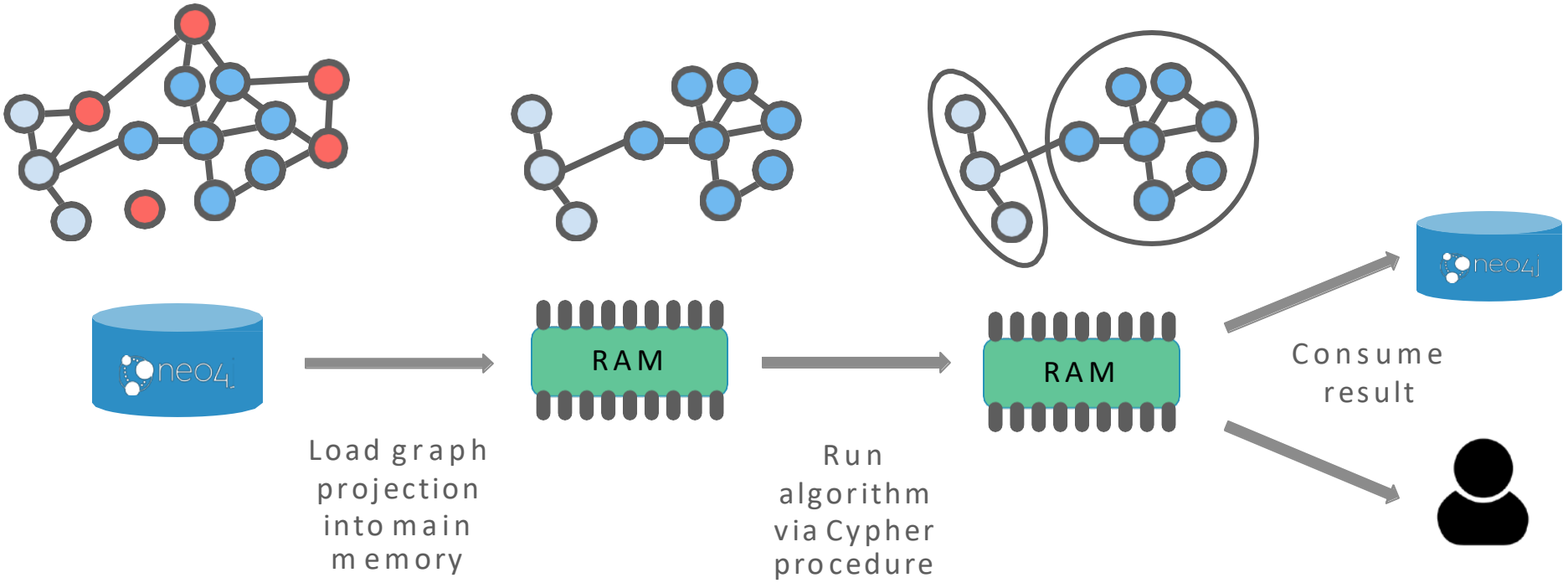


You know what you're looking for and making a decision



You're learning the overall structure of a network, updating data, and predicting

Workflow


RAM
RAM
Load graph projection into main memory
Run algorithm via Cypher procedure
Consume result

Available Algorithms

Community Detection

- **Label Propagation**
- **Louvain**
- **Weakly Connected Components**
- Triangle Count
- Clustering Coefficients
- Strongly Connected Components
- Balanced Triad (identification)

Centrality / Importance

- **PageRank**
- **Personalized PageRank**
- Degree Centrality
- Closeness Centrality
- Betweenness Centrality
- ArticleRank
- Eigenvector Centrality

Similarity

- **Node Similarity**
- Euclidean Distance
- Cosine Similarity
- Overlap Similarity
- Pearson Similarity

Link Prediction

- Adamic Adar
- Common Neighbors
- Preferential Attachment
- Resource Allocations
- Same Community
- Total Neighbors

Pathfinding & Search

- Parallel Breadth First Search
- Parallel Depth First Search
- Shortest Path
- Minimum Spanning Tree
- A* Shortest Path
- Yen's K Shortest Path
- K-Spanning Tree (MST)
- Random Walk

GDS - Algo Syntax

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

Available Modes:

- `stream`: streams results back to the user.
- `write`: writes results to the Neo4j database and returns a summary of the results.
- `stats`: runs the algorithm and only reports statistics.
- `mutate:` writes results to the in-memory graph and returns a summary of the results.

```
CALL gds.wcc.write(  "got-
 interactions",
 {
  writeProperty: "component",
  consecutiveIds: true
 }
) YIELDS writeMillis, componentCount
```

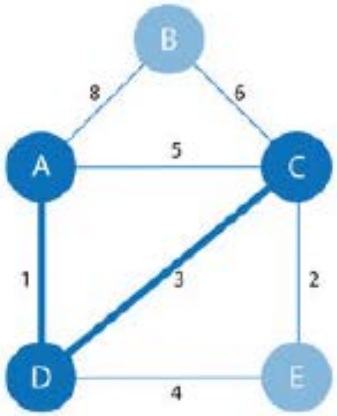
```
CALL gds.wcc.stream("got-
 interactions",
 {}
) YIELDS nodeId, componentId
```

Graph algorithms

- Implemented in the Neo4j Graphs library
- Three main categories:
  - Pathfinding and search
    - Shortest path, minimum spanning trees
  - Centrality computation
    - Node importance
  - Community detection
    - Connectness

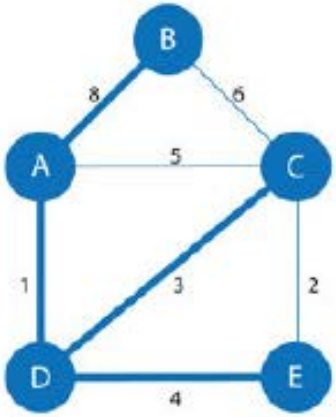
Pathfinding algorithms

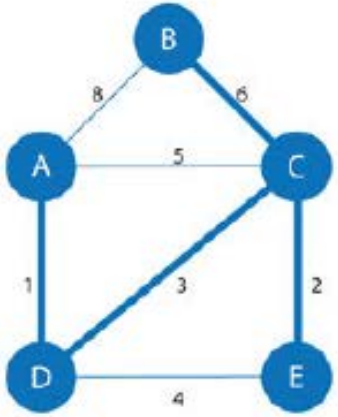
Pathfinding Algorithms



Calculated by relationship weights

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





Shortest Path

Shortest path between 2 nodes (A to C shown)

All-Pairs Shortest Paths

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

Single Source Shortest Path

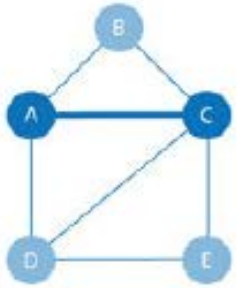
Shortest path from a **root** node (A shown) to **all** other nodes

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

Minimum Spanning Tree

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

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



Calculated by number of hops

Pathfinding Example: The transport Graph

- Graph containing a subset of the European road network

Nodes

id	latitude	longitude	population
Utrecht	52.092876	5.104480	334176
Den Haag	52.078663	4.288788	514861
Immingham	53.61239	-0.22219	9642
Doncaster	53.52285	-1.13116	302400
Hoek van Holland	51.9775	4.13333	9382
Felixstowe	51.96375	1.3511	23689
Ipswich	52.05917	1.15545	133384
Colchester	51.88921	0.90421	104390
London	51.509865	-0.118092	8787892
Rotterdam	51.9225	4.47917	623652
Gouda	52.01667	4.70833	70939

Relationships

src	dst	relationship	cost
Amsterdam	Utrecht	EROAD	46
Amsterdam	Den Haag	EROAD	59
Den Haag	Rotterdam	EROAD	26
Amsterdam	Immingham	EROAD	369
Immingham	Doncaster	EROAD	74
Doncaster	London	EROAD	277
Hoek van Holland	Den Haag	EROAD	27
Felixstowe	Hoek van Holland	EROAD	207
Ipswich	Felixstowe	EROAD	22
Colchester	Ipswich	EROAD	32
London	Colchester	EROAD	106
Gouda	Rotterdam	EROAD	25
Gouda	Utrecht	EROAD	35
Den Haag	Gouda	EROAD	32
Hoek van Holland	Rotterdam	EROAD	33

Import data using cypher (last class)

- We'll start by loading the nodes:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/transport-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (place:Place {id:row.id})
SET place.latitude = toFloat(row.latitude),
place.longitude = toFloat(row.latitude),
place.population = toInteger(row.population)
```

- And now the relationships:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/transport-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (origin:Place {id: row.src})
MATCH (destination:Place {id: row.dst})
MERGE (origin)-[:EROAD {distance: toInteger(row.cost)}]->(destination)
```

Shortest Path (Dijkstra's alg)

- No weights

```
MATCH (source:Place {id: "Amsterdam"}), (destination:Place {id: "London"})
CALL gds.alpha.shortestPath.stream({
startNode: source,endNode: destination,    nodeProjection: "*",
relationshipProjection: {all: { type: "*", properties: "distance",
orientation: "UNDIRECTED“ }}    })
YIELD nodeId, cost
RETURN gds.util.asNode(nodeId).id AS place, cost;
```

- Weighted by distance add relationshipWeightProperty: "distance“ to the relationshipProjection

Minimum Spanning Tree

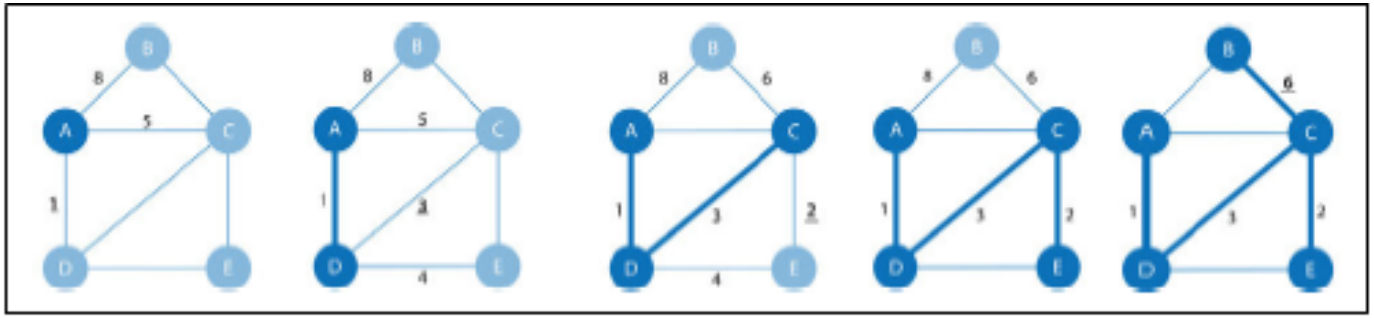


Figure 4-10. The steps of the Minimum Spanning Tree algorithm

```
MATCH (n:Place {id:"Amsterdam"})
CALL gds.alpha.spanningTree.minimum.write({
startNodeId: id(n), nodeProjection: "*",       relationshipProjection: {EROAD: {type: "EROAD",
properties: "distance",
orientation: "UNDIRECTED“   }      },     relationshipWeightProperty: "distance",
writeProperty: 'MINST’,  weightWriteProperty: 'cost’   })
YIELD createMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN createMillis, computeMillis, writeMillis, effectiveNodeCount;
```

Result stored in the graph. Retrieve the minimum weight spanning tree from the graph

Centrality algorithms

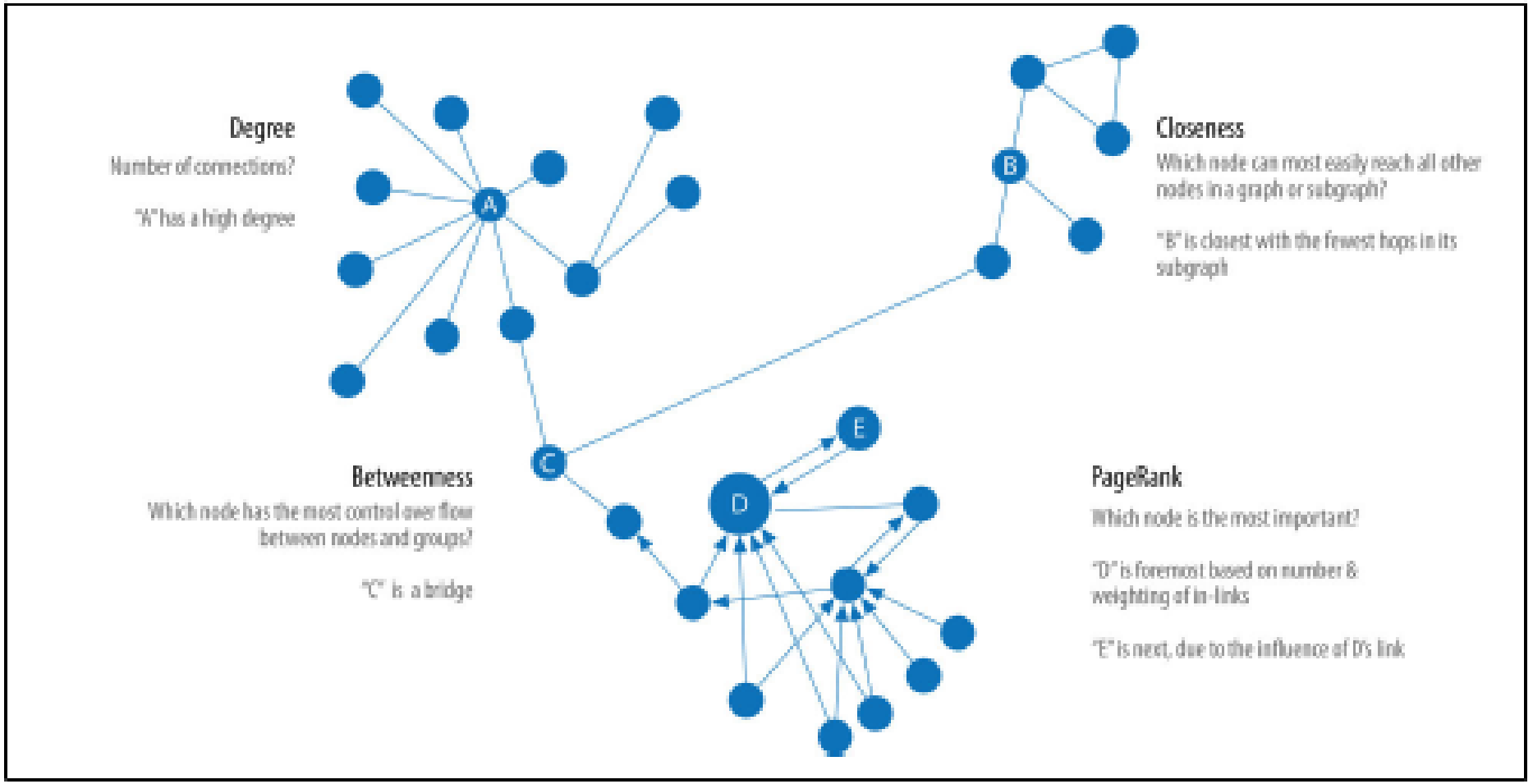


Figure 5-1. Representative centrality algorithms and the types of questions they answer

Centrality Examples: The social graph

- Small Twitter-like graph

id
Alice
Bridget
Charles
Doug
Mark
Michael
David
Amy
James

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

Import social graph

- The following query imports nodes:
- **WITH** "https://github.com/neo4j-graph-analytics/book/raw/master/data/social-nodes.csv" **AS** uri
- LOAD CSV **WITH** HEADERS FROM uri **AS** row
- MERGE (:User {id: row.id});
- And this query imports relationships:
- **WITH** "https://github.com/neo4j-graph-analytics/book/raw/master/data/social-relationships.csv" **AS** uri
- LOAD CSV **WITH** HEADERS FROM uri **AS** row
- **MATCH** (source:User {id: row.src})
- **MATCH** (destination:User {id: row.dst})
- MERGE (source)-[:FOLLOWS]->(destination);

Closeness Centrality

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

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

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

Betweenness Centrality

- A way of detecting the amount of influence a node has over the flow of information or resources in a graph

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

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

PageRank

- PageRank is defined in the original Google paper as follows:

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

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

Social Graph

- MATCH p=(:User)-[:FOLLOWS]->(:User) RETURN p;
- CALL gds.graph.create('nameOfGraph','NodeLabel','RelationshipType’)
- CALL gds.graph.create(‘pagerank_example’, ‘User’, ‘FOLLOWS’);

pageRank Stream

- CALL gds.pageRank.stream('pagerank_example', {maxIterations: 20, dampingFactor: 0.85})
  YIELD nodeId, score
  RETURN gds.util.asNode(nodeId).id AS name, score
  ORDER BY score DESC LIMIT 10

Write back the results

- CALL gds.<algorithm>.write({writeProperty:'pagerank’})
- CALL gds.pageRank.write(‘pagerank_example’, {maxIterations: 20, dampingFactor: 0.85, writeProperty:’pageRank’})

- CALL gds.graph.drop(‘pagerank_example’);

PageRank

- *PageRank is named after Google cofounder Larry Page, who created it to rank websites in Google's search results. The basic assumption is that a page with more incoming and more influential incoming links is more likely a credible source. PageRank measures the number and quality of incoming relationships to a node to determine an estimation of how important that node is. Nodes with more sway over a network are presumed to have more incoming relationships from other influential nodes.*

PageRank

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

Community detection algorithms

Algorithm type	What it does	Example use
Triangle Count and Clustering Coefficient	Measures how many nodes form triangles and the degree to which nodes tend to cluster together	Estimating group stability and whether the network might exhibit "small-world" behaviors seen in graphs with tightly knit clusters
Strongly Connected Components	Finds groups where each node is reachable from every other node in that same group *following the direction* of relationships	Making product recommendations based on group affiliation or similar items
Connected Components	Finds groups where each node is reachable from every other node in that same group, *regardless of the direction* of relationships	Performing fast grouping for other algorithms and identify islands
Label Propagation	Infers clusters by spreading labels based on neighborhood majorities	Understanding consensus in social communities or finding dangerous combinations of possible co-prescribed drugs
Louvain Modularity	Maximizes the presumed accuracy of groupings by comparing relationship weights and densities to a defined estimate or average	In fraud analysis, evaluating whether a group has just a few discrete bad behaviors or is acting as a fraud ring

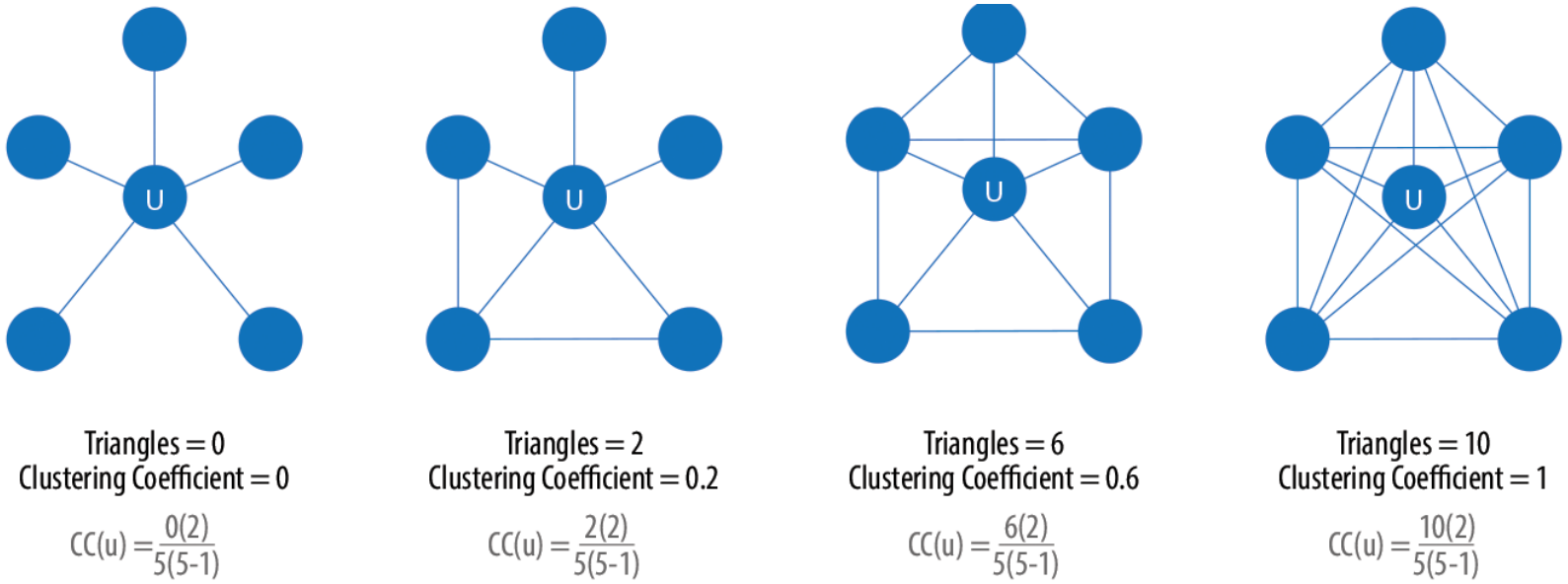
Louvain Example

- https://neo4j.com/docs/graph-data-science/current/algorithms/louvain/

The software graph

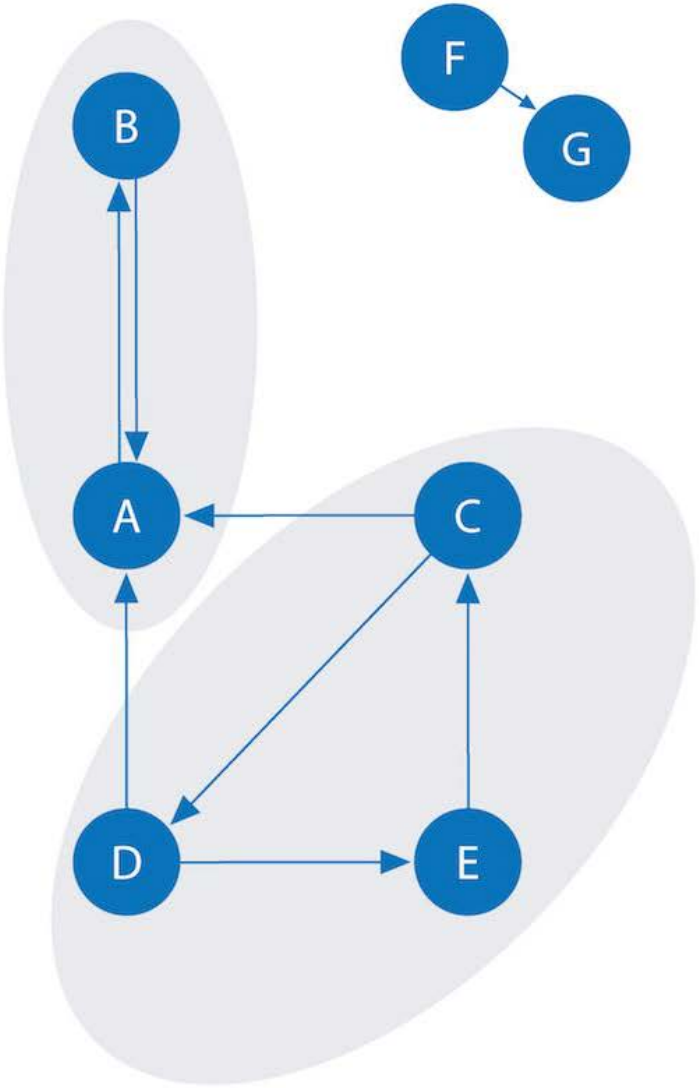
- The following query imports the nodes:
- **WITH** "https://github.com/neo4j-graph-analytics/book/raw/master/data/sw-nodes.csv" **AS** uri
- LOAD CSV **WITH** HEADERS FROM uri **AS** row
- MERGE (:Library {id: row.id});
- And this imports the relationships:
- **WITH** "https://github.com/neo4j-graph-analytics/book/raw/master/data/sw-relationships.csv" **AS** uri
- LOAD CSV **WITH** HEADERS FROM uri **AS** row
- **MATCH** (source:Library {id: row.src})
- **MATCH** (destination:Library {id: row.dst})
- MERGE (source)-[:DEPENDS_ON]->(destination);

Local Clustering Coefficient




```
CALL gds.localClusteringCoefficient.stream({
nodeProjection: "Library",
relationshipProjection: {
DEPENDS_ON: {type: "DEPENDS_ON", orientation: "UNDIRECTED“ } } })
YIELD nodeId, localClusteringCoefficient
WHERE localClusteringCoefficient > 0
RETURN gds.util.asNode(nodeId).id AS library, localClusteringCoefficient
ORDER BY localClusteringCoefficient DESC;
```

Strongly connected components



Strongly Connected Components

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

2 sets of strongly connected components are shown shaded : {A,B,} and {C,D,E}

Note that in {C,D,E} each node can reach the others, but in some cases they must go through another node first.

Connected components

- There exist a path connecting the nodes

```
CALL gds.wcc.stream({
nodeProjection: "Library",
relationshipProjection: "DEPENDS_ON"
})
YIELD nodeId, componentId
RETURN componentId, collect(gds.util.asNode(nodeId).id) AS libraries

ORDER BY size(libraries) DESC;
```

For today…

- To the software dependency graph, add the node naibr with the following dependencies (numpy, scipy, and matplotlib) using the merge command
- Run the label propagation (https://neo4j.com/docs/graph-data-science/current/algorithms/label-propagation/ or Graph Algorithms book page. 139)
- Upload the label propagation results ignoring the direction of the relationship to ICON