Neo4j Graph Data Science

Graph Algorithms

neo4j

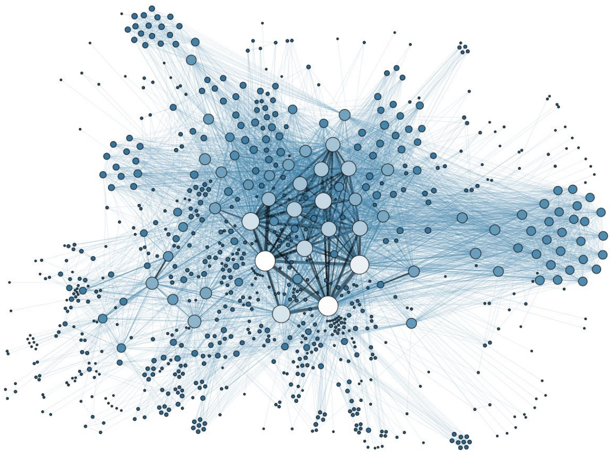
Outline

- What are graph algorithms and how can we use them?
- Algorithms support tiers and execution modes
- Algorithm categories:
  - Centrality
  - Community detection
  - Similarity
  - Path finding
  - Link prediction
- Auxiliary procedures and utility functions
- Best Practices

What *are* graph algorithms?

Is an iterative analysis to make calculations that describe the topology and connectivity of your graph

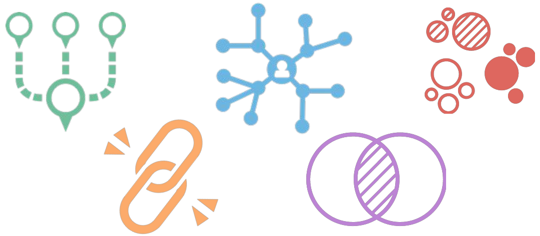
- Generally Unsupervised
- Global traversals & computations
- Learning overall structure
- Typically heuristics and approximations
- Extracting new data from what you already have



How can they be used?

Stand Alone Solution

Find significant patterns and optimal structures



Use community detection and similarity scores for recommendations

Machine Learning Pipeline

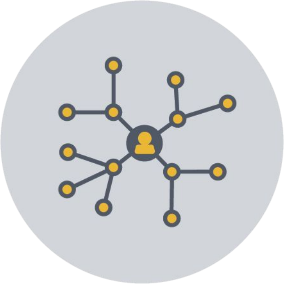
Use the measures as features to train an ML model

1st node	2nd node	Common neighbors	Preferential attachment	Label
1	2	4	15	1
3	4	7	12	1
5	6	1	1	0

Graph Algorithms Categories

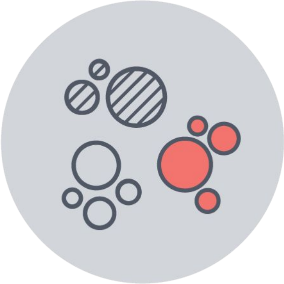
Pathfinding and Search

Finds optimal paths or evaluates route availability and quality.



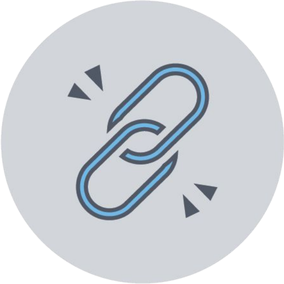
Centrality

Determines the importance of distinct nodes in the network.



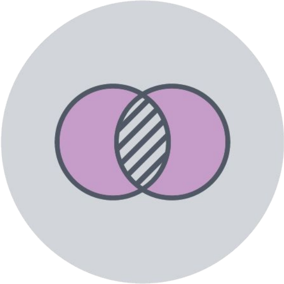
Community Detection

Detects group clustering or partition.



Heuristic Link Prediction

Estimates the likelihood of nodes forming a future relationship.



Similarity

Evaluates how alike nodes are by neighbors and relationships.

Algorithms Per Category

Pathfinding & Search

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

Centrality / Importance

- Degree Centrality
- Closeness Centrality
- CC Variations: Harmonic, Dangalchev, Wasserman & Faust
- Betweenness Centrality
- Approximate Betweenness Centrality
- **PageRank**
- Personalized PageRank
- ArticleRank
- Eigenvector Centrality

Community Detection

- Triangle Count
- Clustering Coefficients
- **Weakly Connected Components**
- Strongly Connected Components
- **Label Propagation**
- **Louvain Modularity**
- K1 coloring
- Modularity optimization

Similarity

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

Link Prediction

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

...and also:

- Random graph generation
- One hot encoding

Tiers of Support

```
CALL gds[.<tier>].<algorithm>.<execution-mode>[.<estimate>](
    graphName: STRING,
    configuration: MAP
)
```

Product supported: Supported by product engineering, tested for stability, scale, fully optimized

```
CALL gds.<algorithm>.<execution-mode>[.<estimate>]
```

Beta: Candidate for product supported tier

```
CALL gds.beta.<algorithm>.<execution-mode>[.<estimate>]
```

Alpha: Experimental implementation, may be changed or removed at any time.

```
CALL gds.alpha.<algorithm>.<execution-mode>[.<estimate>]
```

Execution Modes

```
CALL gds[.<tier>].<algorithm>.<execution-mode>[.<estimate>](
    graphName: STRING,
    configuration: MAP
)
```

Stream: Stream your results back as Cypher result rows. Generally node id(s) and scores.

```
CALL gds[.<tier>].<algorithm>.stream[.<estimate>]
```

Write: Write your results back to Neo4j as node or relationship properties, or new relationships. Must specify `writeProperty`

```
CALL gds[.<tier>].<algorithm>.write[.<estimate>]
```

Mutate: update the in-memory graph with the results of the algorithm

```
CALL gds[.<tier>].<algorithm>.mutate[.<estimate>]
```

Stats: Returns statistics about the algorithm output - percentiles, counts

```
CALL gds[.<tier>].<algorithm>.stats[.<estimate>]
```

Estimation

```
CALL gds[.<tier>].<algorithm>.<execution-mode>[.<estimate>](
    graphName: STRING,
    configuration: MAP
)
```

Estimate lets you estimate the memory requirements for running your algorithm with the specified configuration -- just like `.estimate` with graph catalog operations.

```
CALL gds.<algorithm>.<execution-mode>.estimate
```

Note: *Only* production quality algorithms support `.stats` and `.estimate`

Calling an Algorithm Procedure

Good news! All algorithms in GDS follow the same syntax:

```
CALL gds[.<tier>].<algorithm>.<execution-mode>[.<estimate>](
    graphName: STRING,
    configuration: MAP
)
```

Common Configuration Parameters

```
CALL gds[.<tier>].<algorithm>.<execution-mode>[.<estimate>](
    graphName: STRING,
    configuration: MAP
)
```

Key	Meaning	Default
`concurrency`	How many concurrent threads can be used when executing the algo?	4
`readConcurrency`	How many concurrent threads can be used when reading data?	concurrency
`writeConcurrency`	How many concurrent threads can be used when writing results?	concurrency
`relationshipWeightProperty`	Property containing the weight (must be numeric)	null
`writeProperty`	Property name to write back to	n/a

Centrality

Centrality algorithms

Determines the importance of distinct nodes in the network.

Developed for distinct uses or types of importance.

Centrality Algorithms

Product supported:

- PageRank

Alpha implementations:

- ArticleRank
- Eigenvector Centrality
- Betweenness Centrality
- Closeness Centrality
- Degree Centrality

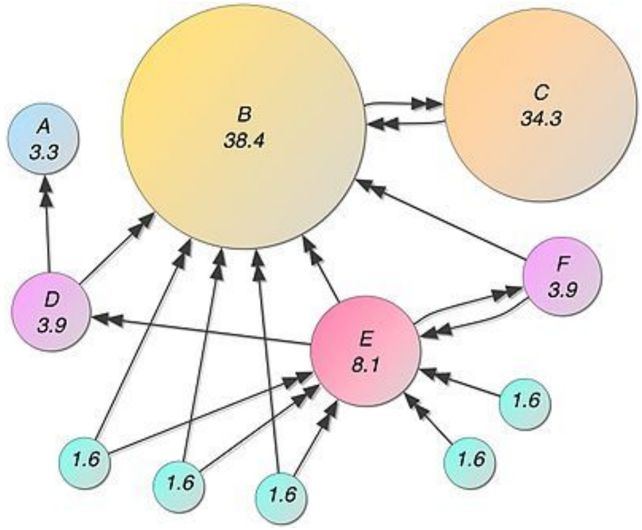
PageRank

What: Finds important nodes based on their relationships

Why: Recommendations, identifying influencers

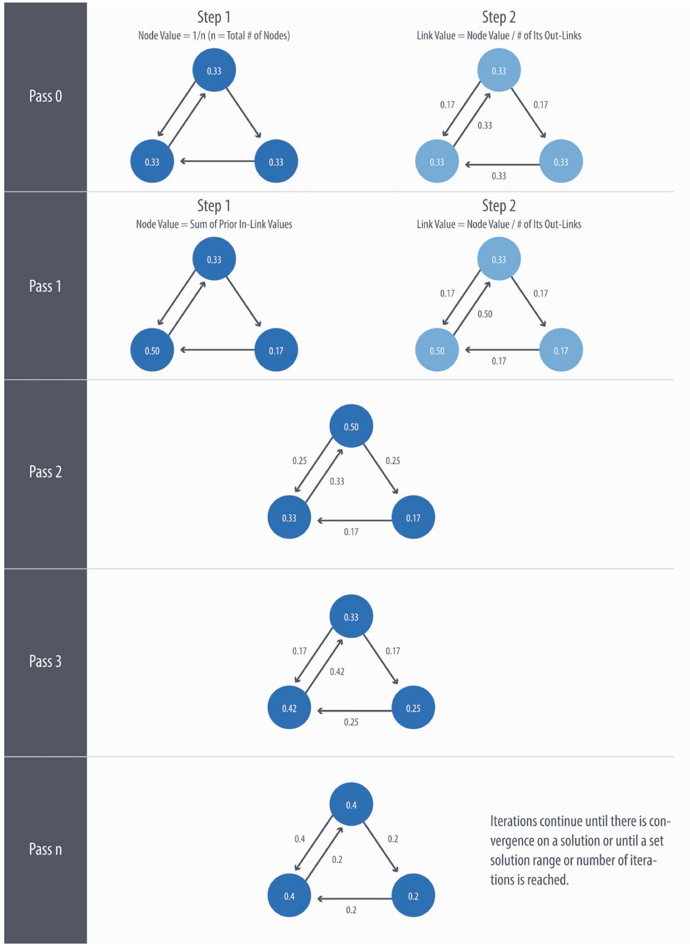
Features:

- Tolerance
- Damping



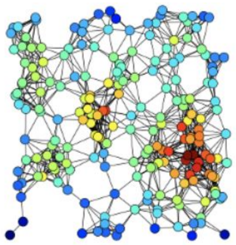
PageRank Calculation

$$PR(A) = (1-d) + d * \left(\frac{PR(T_1)}{C(T_1)} + \ldots + \frac{PR(T_n)}{C(T_n)} \right)$$



Other Centrality Algorithms

Degree Centrality:

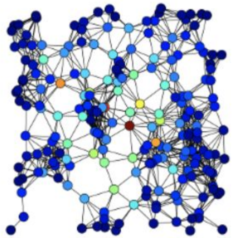


`gds.alpha.degree`

What: Calculated based on the number of relationships.

Why: Outlier identification, preprocessing, influence

Betweenness Centrality:

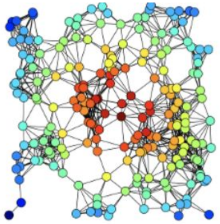


`gds.alpha.betweenness`

What: Calculated based on how often a node acts as a bridge on the shortest path between nodes.

Why: Finding network vulnerabilities, influence

Closeness Centrality:



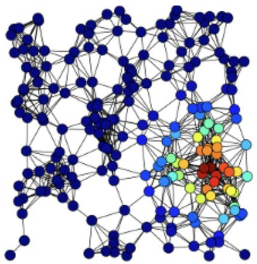
`gds.alpha.closeness`

What: Calculated based on average length of the shortest path to all other nodes.

Why: Diffusing information quickly

Other Centrality Algorithms

Eigenvector Centrality:

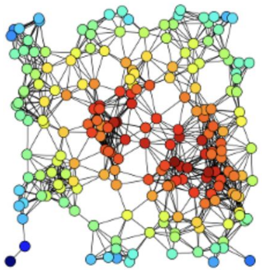


`gds.alpha.eigenvector`

What: PageRank variant, assumes connections to highly connected nodes are more important.

Why: Influence

Harmonic Centrality:



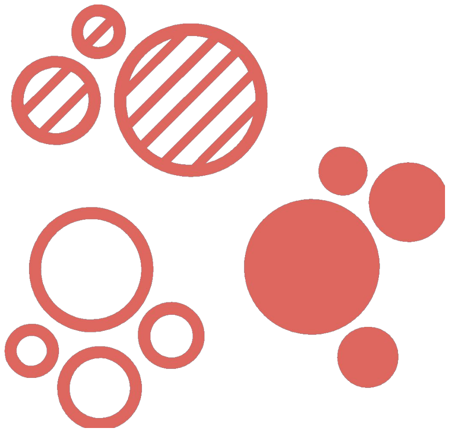
`gds.alpha.harmonic`

What: Closeness centrality variant, reverses sum and reciprocal in formula

Why: Finding network vulnerabilities, influence

Community Detection

Community Detection Algorithms



Evaluates how a group is clustered or partitioned.

Different approaches to define a community.

Community Detection Algorithms

Product supported:

- Weakly Connected Components (UnionFind)
- Label Propagation
- Louvain Modularity

Alpha implementations:

- Strongly Connected Components
- Triangle Counting & Clustering Coefficients

Beta implementations:

- K1 coloring
- Modularity optimization

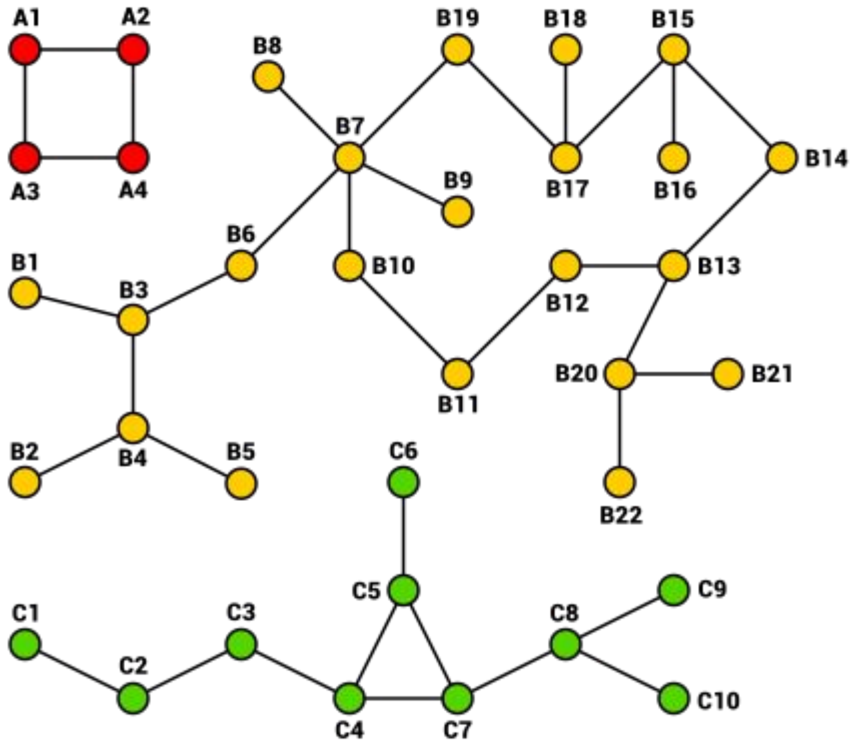
Weakly Connected Components (WCC)

What: Finds disjoint subgraphs in an undirected graph

Why: Graph preprocessing, disambiguation

Features:

- Seeding
- Thresholds
- Consecutive identifiers



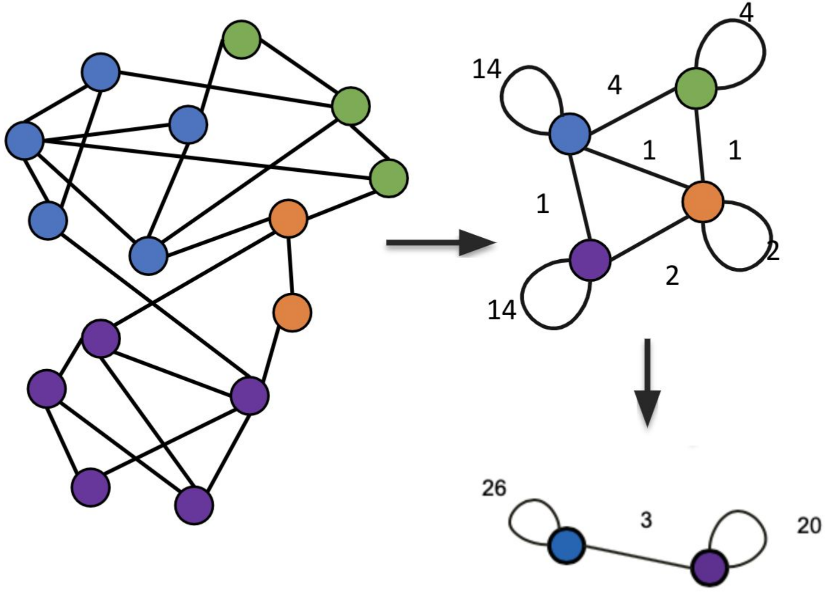
Louvain Modularity

What: Finds communities

Why: Useful for recommendations, fraud detection. *Slower but produces hierarchical results.*

Features:

- Seeding
- Weighted relationships
- Intermediate communities
- Tolerance
- Max Levels, Max Iterations



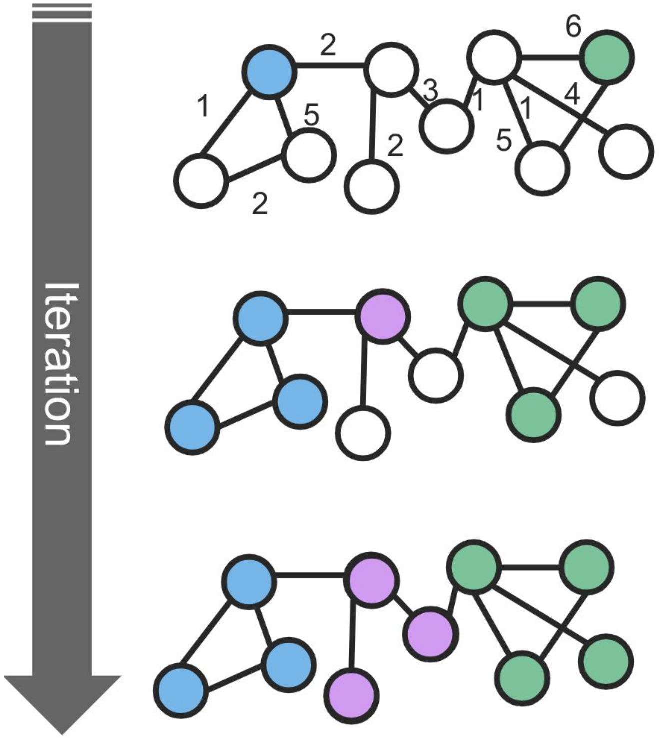
Label Propagation

What: Finds communities

Why: Useful for recommendations, fraud detection, finding common co-occurrences. *Very fast.*

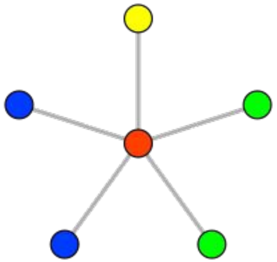
Features:

- Seeding
- Directed relationships
- Weighted relationships



Other community detection algorithms

K1 coloring:

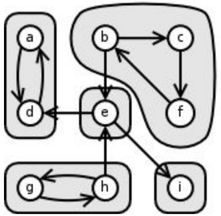


`gds.beta.k1coloring`

What: Find the approximate minimum number of colors so no two adjacent nodes have the same color.

Why: Preprocessing, scheduling optimization.

Strongly connected components

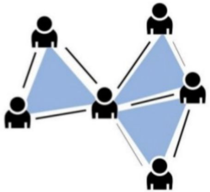


`gds.alpha.scc`

What: Like weakly connected components, but includes directionality.

Why: Finding subgraphs, preprocessing.

Triangle count/ clustering coefficient



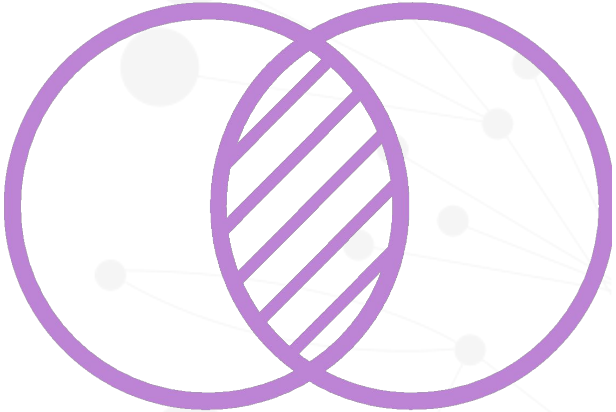
`gds.alpha.triangleCount`

What: Find the number of triangles passing through each node in the graph.

Why: Measuring graph density, stability, and cohesion.

Similarity

Similarity Algorithms



Evaluates how alike nodes are at an individual level either based on node attributes, neighboring nodes, or relationship properties.

Similarity Algorithms

Product supported:

- Node Similarity

Alpha implementations:

- Jaccard
- Cosine
- Pearson
- Euclidean Distance
- Overlap
- Approximate Nearest Neighbors

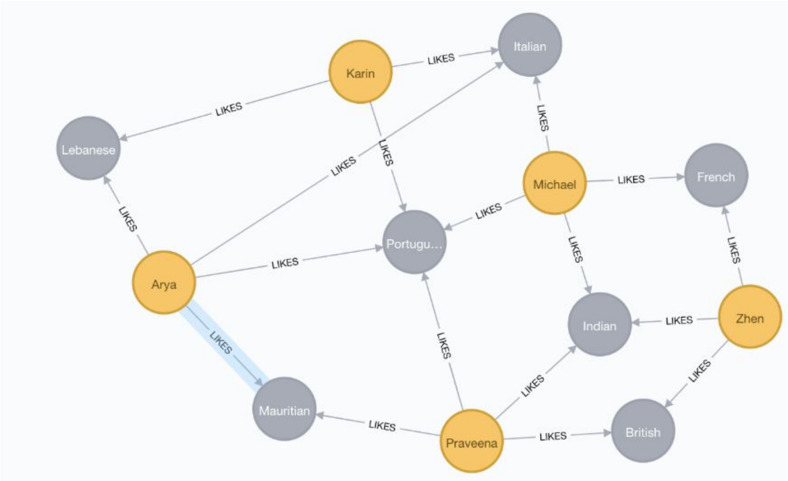
Node Similarity

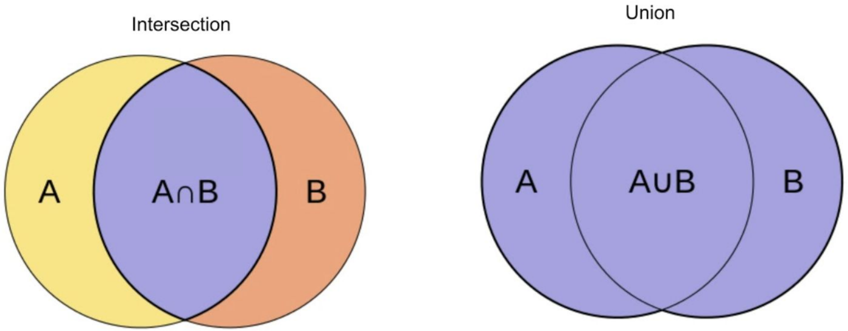
What: Similarity between nodes based on neighbors, intended for bipartite graph. Writes new relationships!

Why: Recommendations, disambiguation

Features:

- topK
- topN
- Degree cutoff
- Similarity cutoff





$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Other Similarity Algorithms

Node Based: These calculate similarity based on common neighbors

- Jaccard
- Overlap similarity

Approximate Nearest Neighbors: This is a faster way of building a similarity graph than Jaccard -- minimizes the number of comparisons

Relationship Based: These calculate the similarity of attributes on the same type of relationship

- Pearson
- Euclidean distance
- Cosine similarity

Link Prediction Function: Common Neighbors

What: The more number of common neighbors between two nodes, the higher the likelihood for a new relationship between them

```
CALL db.relationshipTypes() YIELD
relationshipType as season
MATCH (n1:Character{name:'Daenerys'})
MATCH (n2:Character{name:'Jon'})
RETURN
gds.alpha.linkprediction.commonNeighbors(
n1, n2, {relationshipQuery:season}) AS
score
```

Why: A high score *predicts* that two nodes will form a relationship in the future.

How likely Jon and Daenerys are going to interact in various seasons?

Path Finding

neo4j

Pathfinding and Graph Search algorithms

Pathfinding and Graph Search algorithms are used to identify optimal routes, and they are often a required first step for many other types of analysis.

Path Finding Algorithms

One : One

- **Shortest Path -** `gds.alpha.shortestPath` - *aka Djikstra's algorithm*
- **A* -** `gds.alpha.shortestPath.astar` – *Variant of Djikstra*
- **Yen's K Shortest Paths –** `gds.alpha.kShortestPaths` – *Top K shortest paths*
- **Breadth First Search –** `gds.alpha.bfs` – *Search via breadth first traversal*
- **Depth First Search** - `gds.alpha.dfs` – *Search via depth first traversal*

One : Many

- **Minimum Weight Spanning Tree –** `gds.alpha.spanningTree` – *Reachable nodes from start node and the minimum weighted relationships between them*
- **K-minimum Weight Spanning Tree –** `gds.alpha.spanningTree.kmin` – *MWST with depth limitation*
- **Single Source Shortest Path –** `gds.alpa.shortestPath.deltaStepping`- *Shortest path from source node to all others*

Many : Many

- **All Pairs Shortest Path -** `gds.alpha.allShortestPaths` – *Shortest path between every pair of nodes*

One : ???

- **Random Walk -** `gds.alpha.randomWalk` – *Random Traversal from a specified start node*

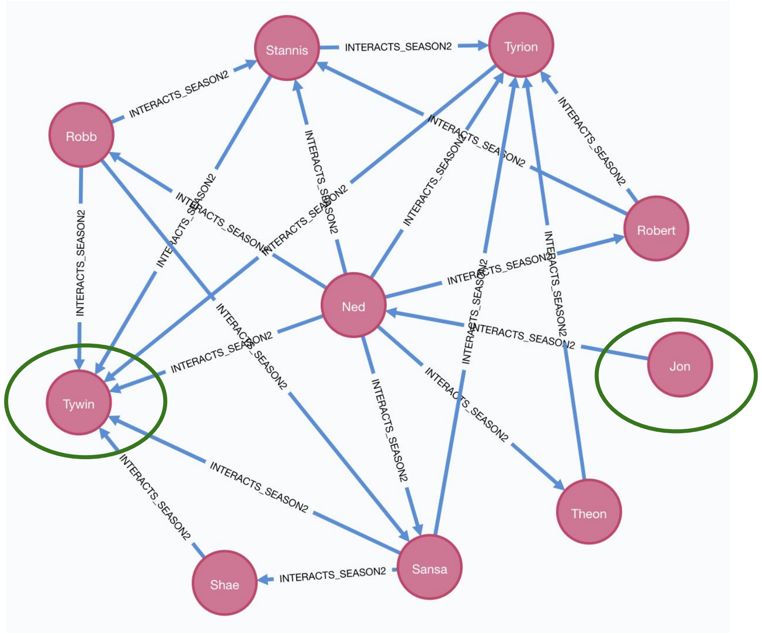
Shortest Path

What: Using Dijkstra's algorithm, find the shortest paths between a source and target node

Why: Degrees of separation between people, directions between physical locations

Features:

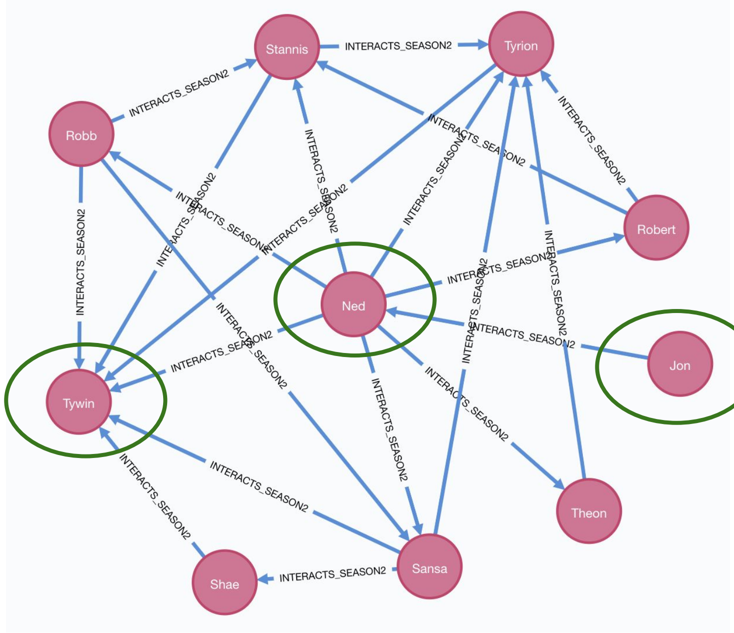
- relationshipWeightProperty



What is the undirected shortest path between Jon and Tywin?

Shortest Path

```
MATCH(start:Person{name:'Jon Snow'})
MATCH(end:Person{name:'Tywin Lannister'})
CALL gds.alpha.shortestPath.stream({
     nodeProjection: Person,
     relationshipProjection:{
        INTERACTS_SEASON2: {
          type: 'INTERACTS_2',
          orientation: 'UNDIRECTED'
        }},
     startNode: start,
     endNode: end
})
YIELD nodeId,cost
RETURN gds.util.asNode(nodeId).name AS
name,cost;
```



['Jon'], ['Ned'], ['Tywin']

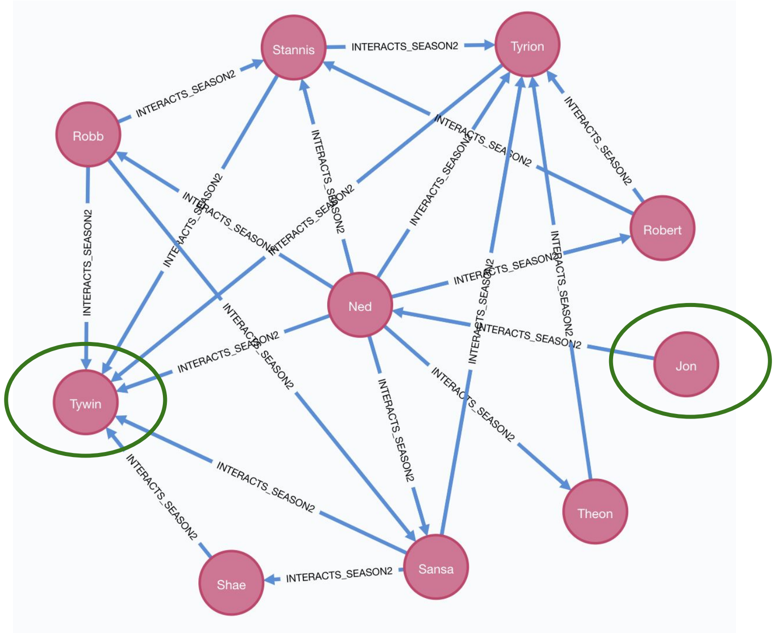
Yen’s K Shortest Paths

What: Using Dijkstra’s algorithm, find the *k* shortest paths between a source and target node

Why: Route availability, assessing network redundancy or resilience

Features:

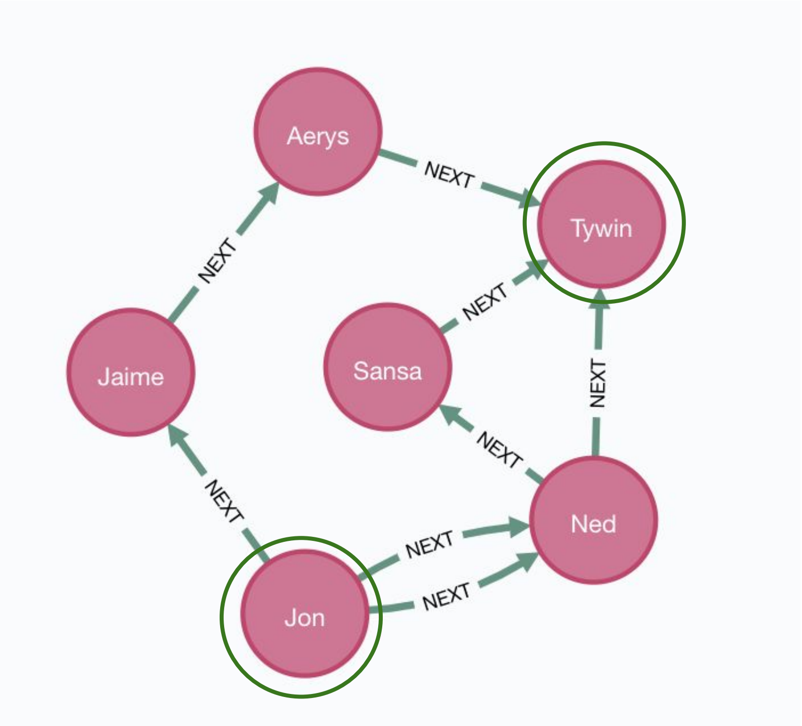
- relationshipWeightProperty
- k (number of paths)
- path



What are the three shortest paths between Jon and Tywin?

Yen's K Shortest Paths Execution Call

```
MATCH(start:Person{name:'Jon Snow'}),
(end:Person{name:'Tywin Lannister'})
CALL gds.alpha.kShortestPaths.stream({
     nodeProjection: 'Person',
     relationshipProjection:{
        INTERACTS_SEASON2: {
          type: 'INTERACTS_2',
         orientation: 'UNDIRECTED',
          properties:'weight'
        }},
     startNode: start,
     endNode: end,
     k: 3,
     relationshipWeightProperty: 'weight',
     path: true
})
YIELD path
RETURN path
```



Question

This syntax looks a little different than the other algorithms 🤔

Can you identify the major differences?

Yen's K Shortest Paths Execution Call

```
MATCH(start:Person{name:'Jon Snow'}),
(end:Person{name:'Tywin Lannister'})
CALL gds.alpha.kShortestPaths.stream({
     nodeProjection: 'Person',
     relationshipProjection:{
        INTERACTS_SEASON2: {
          type: 'INTERACTS_2',
         orientation: 'UNDIRECTED',
          properties:'weight'
        }},
     startNode: start,
     endNode: end,
     k: 3,
     relationshipWeightProperty: 'weight',
     path: true
})
YIELD path
RETURN path
```

We're using an **anonymous graph** instead of a pre-loaded named graph

Yen’s K Shortest Paths Execution Call

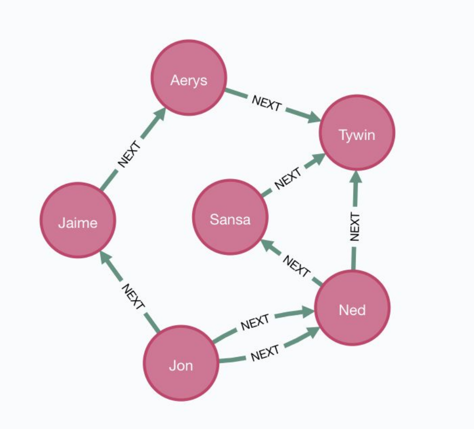
```
MATCH(start:Person{name:'Jon Snow'}),
(end:Person{name:'Tywin Lannister'})
CALL gds.alpha.kShortestPaths.stream({
     nodeProjection: 'Person',
     relationshipProjection:{
        INTERACTS_SEASON2: {
          type: 'INTERACTS_2',
         orientation: 'UNDIRECTED',
          properties:'weight'
        }},
     startNode: start,
     endNode: end,
     k: 3,
     relationshipWeightProperty: 'weight',
     path: true
})
YIELD path
RETURN path
```

You have to specify the start and end nodes -- first identifying them with a Cypher `MATCH` statement, then referring to them in the algo call

Yen's K Shortest Paths Execution Call

```
MATCH(start:Person{name:'Jon Snow'}),
(end:Person{name:'Tywin Lannister'})
CALL gds.alpha.kShortestPaths.stream({
    nodeProjection: 'Person',
    relationshipProjection:{
        INTERACTS_SEASON2: {
          type: 'INTERACTS_2',
         orientation: 'UNDIRECTED',
          properties:'weight'
        }},
    startNode: start,
    endNode: end,
    k: 3,
    relationshipWeightProperty: 'weight',
    path: true
})
YIELD path
RETURN path
```

This algorithm can return a path, or paths, if you set path to true!



Link Prediction

neo4j

Link Prediction

These methods compute a score for a pair of nodes, where the score could be considered a **measure of proximity** or “similarity” between those nodes **based on the graph topology**.

Link Prediction Functions

Link prediction algorithms are **functions** which

(1) take a pair of nodes as input and
(2) use a formula to calculate the probability that there *should* be an edge between them

```
MATCH (n1:Node {id:'id1'})
MATCH (n2:Node {id:'id2'})
RETURN gds.alpha.linkPrediction.<algoName>(n1,n2,
     {relationshipQuery:'relationshipType') AS score
```

Where the available algorithms are: adamic adar, common neighbors, preferential attachment, resource allocation, same community, and total neighbors

Link Prediction Function: Preferential Attachment

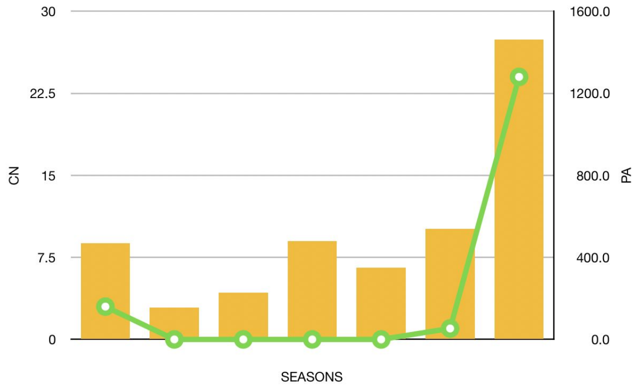
What: The better connected a node is, the more likely it is to form new edges (think: popular people make more friends)

Why: A high preferential attachment score *predicts* that two nodes will form a relationship in the future.

$$PA(x, y) = |N(x)| * |N(y)|$$

```
CALL db.relationshipTypes() YIELD
relationshipType as season
MATCH (n1:Person{name:'Daenerys
Targaryen'})
MATCH (n2:Person{name:'Jon Snow'})
RETURN
gds.alpha.linkprediction.preferentialAttac
hment(n1, n2, {relationshipQuery:season})
AS score
```

How likely Jon and Daenerys are going to interact in various seasons?



Auxiliary Procedures & Utility Functions

Auxiliary Procedures

Some procedures don't quite fit into the other categories... but are still useful!

- **Graph Generation**: Make an in-memory graph with a set number of nodes and relationships, which follow a specific degree distribution.

  ```
  CALL gds.beta.graph.generate
  ```

  This lets you approximate a graph and estimate run times on their hardware and set up.
- **One hot encoding**: Convert labels into vectors -- useful for ML

  ```
  CALL gds.alpha.ml.oneHotEncoding
  ```

Utility Functions

The GDS library includes a number of functions that help pre- and post-process data to fit into your cypher based workflows:

Pre-processing:

- `gds.util.NaN`
- `gds.util.isFinite`, `gds.util.isInfinite`
- `gds.util.infinity`

Post-processing:

- `gds.util.asNode`, `gds.util.asNodes`
- `gds.util.asPath` *warning! this does not return a **real** path!*

Helpers:

- `gds.graph.exists`
- `gds.version`

Best Practices

neo4j

Recommended Steps