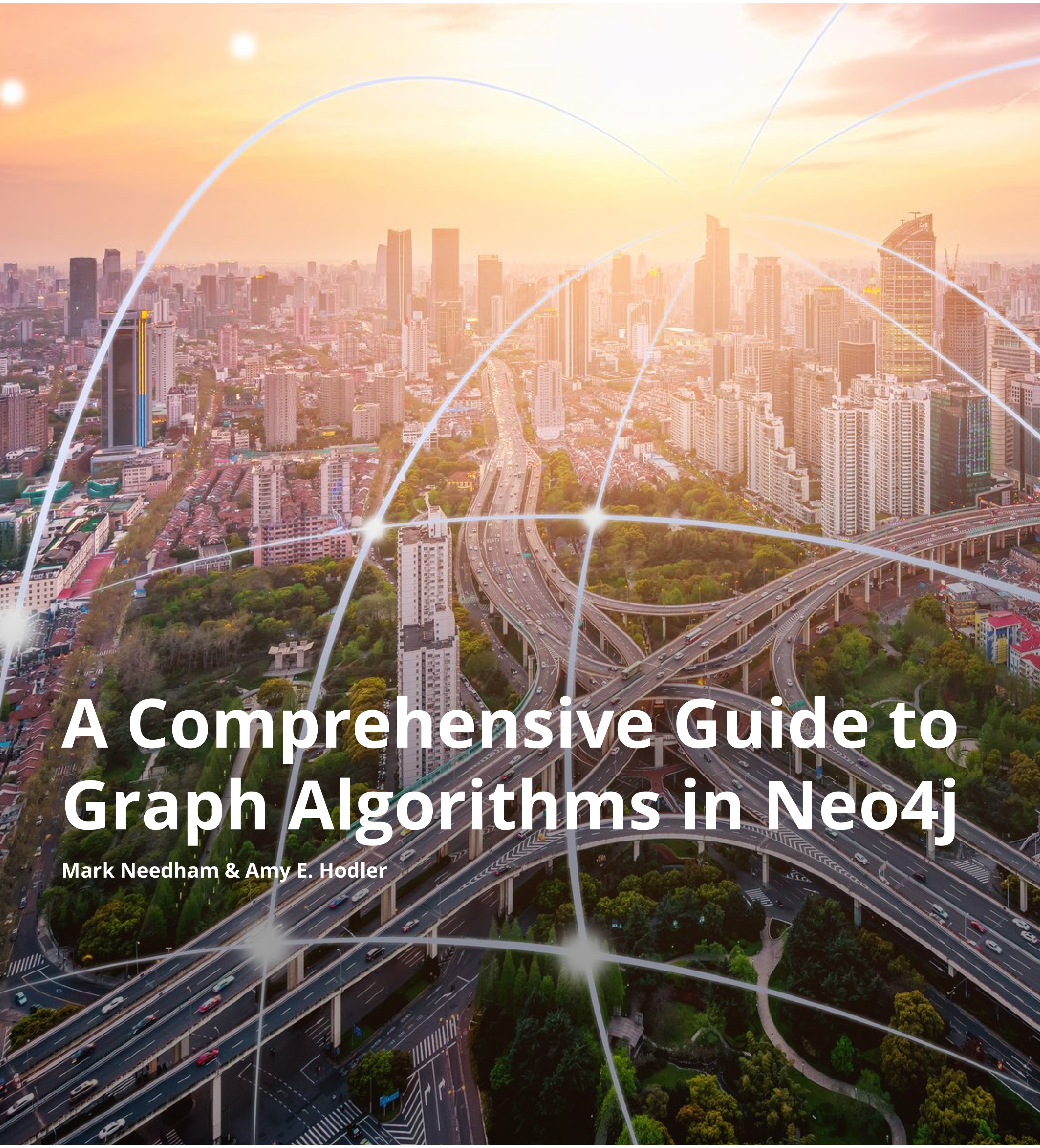
The #1 Platform for Connected Data



A Comprehensive Guide to Graph Algorithms in Neo4j

Mark Needham & Amy E. Hodler

neo4j.com

Ebook

TABLE OF CONTENTS

A Comprehensive Guide to Graph Algorithms

Mark Needham, Developer Relations Engineer
Amy E. Hodler, Director, Graph Analytics and AI Programs

Preface

Connectivity is the single most pervasive characteristic of today's networks and systems.

From protein interactions to social networks, from communication systems to power grids, and from retail experiences to supply chains – networks with even a modest degree of complexity are not random, which means connections are not evenly distributed nor static. This is why simple statistical analysis alone fails to sufficiently describe – let alone predict – behaviors within connected systems. Consequently, most big data analytics today do not adequately model the connectedness of real-world systems and have fallen short in extracting value from huge volumes of interrelated data.

As the world becomes increasingly interconnected and systems increasingly complex, it's imperative that we use technologies built to leverage relationships and their dynamic characteristics. Not surprisingly, interest in graph data science has exploded because it was explicitly developed to gain insights from connected data. Graph analytics reveal the workings of intricate systems and networks at massive scales – not only for large labs but for any organization. Graph algorithms are processes used to run calculations based on mathematics specifically created for connected information.

We are passionate about the utility and importance of graph analytics as well as the joy of uncovering the inner workings of complex scenarios. Until recently, adopting graph analytics required significant expertise and determination, since tools and integrations were difficult and few knew how to apply graph algorithms to their quandaries. It is our goal to help change this. We wrote this ebook to help organizations better leverage graph analytics so they make new discoveries and develop intelligent solutions faster.

While there are other graph algorithm libraries and solutions, we've chosen to focus on the graph algorithms in the Neo4j platform. However, you'll find this guide helpful for understanding more general graph concepts regardless of what graph technology you use.

"Graph analysis is possibly the single most effective competitive differentiator for organizations pursuing data-driven operations and decisions."

– *Gartner Research*

How to Use This Ebook

This ebook is written in two parts. For product managers and solution owners, Part I provides an overview of graph algorithms and their uses. In these chapters, the background of graph analytics is used to illustrate basic concepts and their relevance to the modern data landscape.

Part II, the bulk of this ebook, is written as a practical guide to getting started with graph algorithms for engineers and data scientists who have some Neo4j experience. It serves as a detailed reference for using graph algorithms. At the beginning of each category of algorithms, there is a reference table to help you quickly jump to the relevant algorithm.

For each algorithm, you'll find:

- An explanation of what the algorithm does
- Use cases for the algorithm and references to read more about them
- Walkthroughs with example code providing concrete ways to use the algorithm

In the reference section, you'll find notes, tips and code.

Note: Details about the workings of the algorithm that you may want to know about.

Tip: Details you should be aware of with regard to the algorithm, such as the types of graphs it works best with or values that are not permitted.

Code examples, node names and relationships are shown in a code font, `Courier New`.

If you have any questions or need any help with any of the material in this ebook, send us an email at devrel@neo4j.com.

Acknowledgments

We've thoroughly enjoyed putting together the material for this ebook and would like to thank all those who assisted. We'd especially like to thank Michael Hunger for his guidance and Tomaz Bratanic for his keen research. Finally, we greatly appreciate Yelp for permitting us to use its rich dataset for powerful examples and Tomer Elmalem for brainstorming with us on ideas.

Part I: Connected Data and Graph Analysis

Chapter 1
Making Sense of Connected Data

The Latin root of valence is the same as value, *valere*, which means to be strong, powerful, influential or healthy.

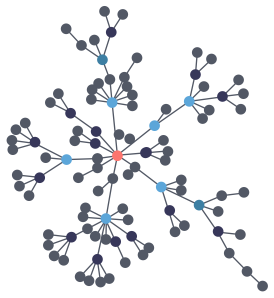
Connected Data Today

There are four to five "Vs" often used to help define big data (volume, velocity, variety, veracity and sometimes value) and yet there's almost always one powerful "V" missing: ***valence***. In chemistry, valence is the combining power of an element; in psychology, it is the intrinsic attractiveness of an object; and in linguistics, it's the number of elements a word combines.

Although valence has a specific meaning in certain disciplines, in almost all cases there is an element of connection and behavior within a larger system. In the context of big data, valence is the tendency of individual data to connect as well as the overall connectedness of datasets. Some researchers measure the valence of a data collection as the ratio of connections to the total number of possible connections. The more connections within your dataset, the higher its valence.

Your data wants to connect, to form new data aggregations and subsets, and then connect to more data and so forth. Moreover, data doesn't arbitrarily connect for its own sake; there's significance behind every connection it makes. In turn, this means that the meaning behind every connection is decipherable after the fact. Although this may sound like something that's mainly applicable in a biological context, most complex systems exhibit this tendency. In fact, we can see this in our daily lives with a simple example of highly targeted purchase recommendations based on the connections between our browsing history, shopping habits, demographics, and even current location. Big data has valence – and it's strong.

Scientists have observed the growth of networks and the relationships within them for some time. Yet there is still much to understand and active work underway to further quantify and uncover the dynamics behind this growth. What we do know is that valence increases over time but not uniformly. Scientists have described preferential attachment (for example, the rich get richer) as leading to power-law distributions and scale-free networks with hub and spoke structures.



Preferential attachment means that the more connected a node is, the more likely it is to receive new links. Source: Wikipedia

Highly dense and lumpy data networks tend to develop, in effect growing both your big data and its complexity. This is significant because densely yet unevenly connected data is very difficult to unpack and explore with traditional analytics. In addition, more sophisticated methods are required to model scenarios that make predictions about a network's evolution over time such as how transportation systems grow. These dynamics further complicate monitoring for sudden changes and bursts, as well as discovering emergent properties. For example, as density increases in a social group, you might see accelerated communication that then leads to a tipping point of coordination and a subsequent coalition or, alternatively, subgroup formation and polarization.

This data-begets-data cycle may sound intimidating, but the emergent behavior and patterns of these connections reveal more about dynamics than you learn by studying individual elements themselves. For example, you could study the movements of a single starling but until you understood how these birds interact with each other in a larger group, you wouldn't understand the dynamics of a flock of starlings in flight. In business you might be able to make an accurate restaurant recommendation for an individual, but it's a significant challenge to estimate the best group activity for seven friends with different dietary preferences and relationship statuses. Ironically, it's this vigorous connectedness that uncovers the hidden value within your data.

Economist Jeffrey Goldstein defined emergence as "the arising of novel and coherent structures, patterns and properties during the process of self-organization in complex systems."

Economist Jeffrey Goldstein defined emergence as "the arising of novel and coherent structures, patterns and properties during the process of self-organization in complex systems." That includes the common characteristics of:

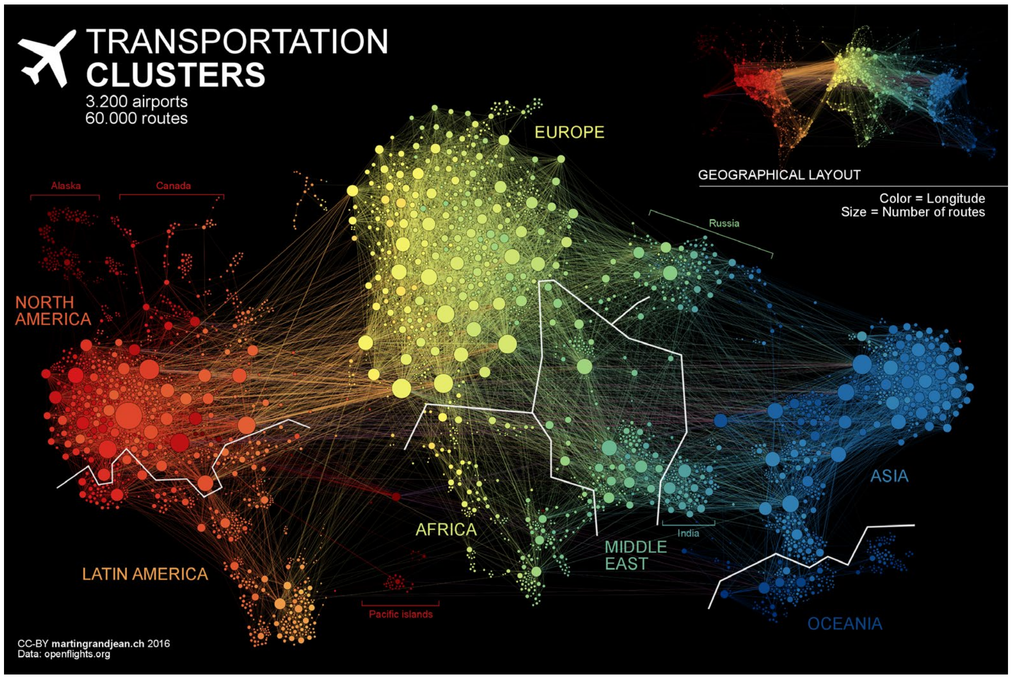
- Radical novelty (features not previously observed in systems);
- Coherence or correlation (meaning integrated wholes that maintain themselves over some period of time);
- A global or macro "level" (i.e., there is some property of "wholeness");
- Being the product of a dynamical process (it evolves); and
- An ostensive nature (it can be perceived). (Source: Wikipedia)

For today's connected data, it's a mistake to scrutinize data elements and aggregations for insights using only simple statistical tools because they make data look uniform and they hide evolving dynamics. Relationships between data are the linchpin of understanding real-world behaviors within – and of – networks and systems.

Network Science & the Rise of Graph Models

Networks are a representation, a tool to understand complex systems and the complex connections inherent in today's data. For example, you can represent how a social system works by thinking about interactions between pairs of people. By analyzing the structure of this representation, we answer questions and make predictions about how the system works or how individuals behave within it. In this sense, network science is a set of technical tools applicable to nearly any domain, and graphs are the mathematical models used to perform analysis.

Networks also act as a bridge for understanding how microscopic interactions and dynamics lead to global or macroscopic regularities as well as correlate small scale clusters to a larger scale element and shape projection. Networks bridge between the micro and the macro because they represent exactly which things are interacting with each other. It's a common assumption that the average of a system is sufficient because the results will even out. However, that's not true. For example, in a social setting, some people interact heavily with others while some only interact with a few. An averages approach to data completely ignores the uneven distributions and locality within real-world networks.



Transportation networks illustrate the uneven distribution of relationships and groupings. Source: Martin Grandjean

An extremely important effort in network science is figuring out how the structure of a network shapes the dynamics of the whole system. Over the last 15 years we've learned that for many complex systems, the network is important in shaping both what happens to individuals within the network and how the whole system evolves.

Graph analytics, based on the specific mathematics of graph theory, examine the overall nature of networks and complex systems through their connections. With this approach, we understand the structure of connected systems and model their processes to reveal hard-to-find yet essential information: propagation pathways, such as the route of diseases or network failures; flow capacity and dynamics of resources, such as information or electricity; or the overall robustness of a system. Understanding networks and the connections within them offers immense potential for breakthroughs by unpacking structures and revealing patterns used for science and business innovations as well as for safeguarding against vulnerabilities, especially those unforeseen within the labyrinth.

Propagation Pathways
Interactions & Resiliency
Flow & Dynamics

The Power of Graph Algorithms

Researchers have found common underlying principles and structures across a wide variety of networks and have figured out how to apply existing, standard mathematical tools (i.e., graph theory) across different network domains.

But this raises questions: How do people who are not mathematicians conversant in network science apply graph analytics appropriately? How can everyone learn from connected data across domains and use cases?

This is where graph algorithms come into play. In the simplest terms, graph algorithms are mathematical recipes based on graph theory that analyze the relationships in connected data.

Even a single graph algorithm has many applications across multiple use cases. For example, the PageRank graph algorithm – invented by Google founder Larry Page – is useful beyond organizing web search results. It's also been used to study the role of species in food webs, to research telomeres in the brain, and to model the influence of particular network components in just about every industry.

In studying the brain, scientists found that the lower the PageRank of a telomere, the shorter it was – and there's a strong correlation between short telomeres and cellular aging.

Graph algorithms play a powerful role in graph analytics, and the purpose of this ebook is to showcase that role. But first let's step back and look at the rise of graph analytics as a whole and its many applications in exploring connected data.

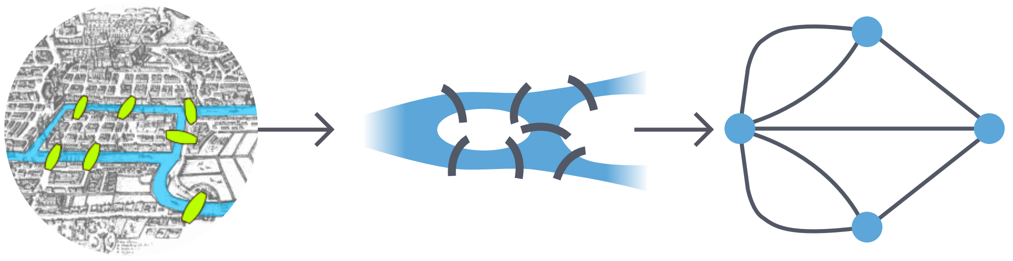
"The tools of graph theory can be utilized in order to analyze the networks and obtain a better understanding of their overall construction. This approach has led to several groundbreaking discoveries on the nature of networks, crossing fields of research from biology, to social science and technology."

– Albert-László Barabási
Director, Center for Complex Network Research, Northeastern University

Chapter 2
The Rise of Graph Analytics

The Roots of Graph Analytics

Graph analytics has a history dating back to 1736, when Leonhard Euler solved the "Seven Bridges of Königsberg" problem. The problem asked whether it was possible to visit all four areas of a city, connected by seven bridges, while only crossing each bridge once. It wasn't. With the insight that only the connections themselves were relevant, Euler set the groundwork for graph theory and its mathematics.



Source: Wikipedia

But graph analytics did not catch on immediately. Two hundred years would pass before the first graph textbook was published in 1936. In the late 1960s and 1970s, network science and applied graph analytics really began to emerge.

In the last few years, there's been an explosion of interest in and usage of graph technologies. Demand is accelerating based on a need to better understand real-world networks and forecast their behaviors, which is resulting in many new graph-based solutions.

Why Now? Forces Fueling the Rise in Graph Analytics

This growth in network science and graph analytics is the result of a combined shift in technical abilities, new insights, and the realization that existing business intelligence systems and simple statistics fail to provide a complete picture of real-world networks. Several forces are driving the rise in graph analytics.

First of all, we've seen real-world applications of graph analytics and their impact on us all. The power of connected data for business benefit has been demonstrated in disruptive success stories such as Google, LinkedIn, Uber, and eBay, among many others.

At the same time, digitization and the growth in computing power (and connected computing) have given us an unprecedented ability to collect, share and analyze massive amounts of data. But despite the masses of data they have, organizations are frustrated with the unfulfilled promises of big data and their inability to analyze it.

The majority of analytics used today handle specific, well-crafted questions efficiently but fall short in helping us predict the behavior of real systems, groups and networks. Most networks defy averages and respond nonlinearly to changes. As a result, more businesses are turning to graph analytics, which are built for connected data and responsive to dynamic changes.

In addition, there's been a recognition of how graphs enhance machine learning and provide a decision-making framework for artificial intelligence. From data cleansing for machine learning to feature extraction in model development to knowledge graphs that provide rich context for AI, graph technology is enhancing AI solutions. This is described in more detail later in this chapter.

Bringing Together Analytics & Transactions

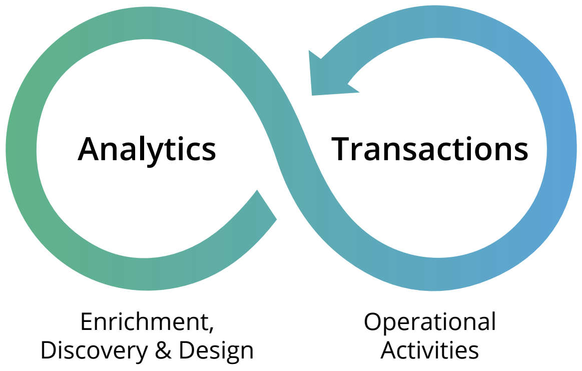
Historically, the worlds of analytics (OLAP) and transactions (OLTP) have been siloed despite their interdependence (analytics drives smarter transactions, which creates new opportunities for analysis), which is especially true with connected data.

This line has been blurred in recent years and modern data-intensive applications combine real-time transactional queries with less time-sensitive analytics queries. The merging of analytics and transactions enables continual analysis to become ingrained in regular operations. As data is gathered – from point-of-sale (POS) systems, from manufacturing equipment, from IoT devices, or from wherever – analytics at the moment and location support an application's ability to make real-time recommendations and decisions. This blending of analytics and transactions was observed several years ago, and terms to describe this blurring and integration include "Transalytics" and Hybrid Transactional and Analytical Processing (HTAP).

"[HTAP] could potentially redefine the way some business processes are executed, as real-time advanced analytics (for example, planning, forecasting and what-if analysis) becomes an integral part of the process itself, rather than a separate activity performed after the fact. This would enable new forms of real-time business-driven decision-making process. Ultimately, HTAP will become a key enabling architecture for intelligent business operations."

– *Gartner*

Graph algorithms provide the means to understand, model and predict complicated dynamics such as the flow of resources or information, the pathways through which contagions or network failures spread, and the influences on and resiliency of groups. Neo4j brings together analytics and transactional operations in a native graph platform, helping not only uncover the inner nature of real-world systems for new discoveries, but also enabling faster development and deployment of graph-based solutions with more closely integrated processing for transactions and analytics.



According to Gartner's Magic Quadrant survey, the biggest reason for using the Neo4j Graph Platform "is to drive innovation."

"We need to combine transactional and analytic systems into transalytic systems and stop thinking about these as two separate systems. 2018 is going to be the year we'll see major corporations collapse these two systems together, so that you have simplified architecture and can move at the pace of business."

– *Bill Powell, Director of Enterprise Architecture, Automotive Resources International (ARI)*

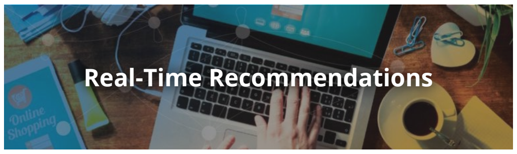
Use Cases for Graph Transactions & Analytics

Today's most pressing data challenges center around connections, not just tabulating discrete data. Graph analytics accelerate breakthroughs across industries with more intelligent solutions.

eBay uses graphs to deliver real-time, personalized user experiences and recommendations. Cybersecurity and fraud systems correlate network, social and IoT data to uncover patterns. More accurate modeling and decisioning for a range of dynamic networks drives use cases from subsecond packaging of financial commodities and routing logistics to IT service assurance to predicting the spread of epidemics. Graph technologies help businesses with many practical use cases across industries and domains, a few of which are highlighted in the sections that follow.

Real-Time Fraud Detection

Traditional fraud prevention measures focus on discrete data points such as specific account balances, money transfers, transaction streams, individuals, devices or IP addresses. However, today's sophisticated fraudsters escape detection by forming fraud rings comprised of stolen and synthetic identities. To uncover such fraud rings, it is essential to look beyond individual data points to the connections that link them. Connections are key to identifying and stopping fraud rings and their ever-shifting patterns of activities. Graph analytics enable us to find these patterns and shows us that indeed, fraud has a shape.



Real-Time Recommendations

Graph-powered recommendation engines help companies personalize products, content and services by contextualizing a multitude of connections in real time. Making relevant recommendations in real time requires the ability to correlate product, customer, historic preferences and attributes, inventory, supplier, logistics and even social sentiment data. Moreover, a real-time recommendation engine requires the ability to instantly capture any new interests shown during the customer's current visit – something that batch processing can't accomplish.

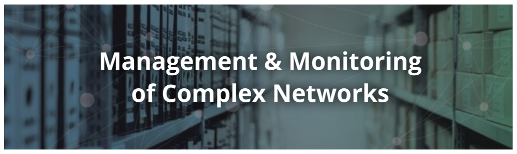
360° View of Data

As businesses become more customer centric, it has never been more urgent to tap the connections in your data to make timely operational decisions. This requires a technology to unify your master data, including customer, product, supplier and logistics information to power the next generation of ecommerce, supply chain and logistics applications.

Organizations gain transformative real-time business insights from relationships in master data when storing and modeling data as a graph. This translates to highlighting time- and cost-saving queries around data ownership, customer experience and support, organizational hierarchies, human capital management, and supply chain transparency.

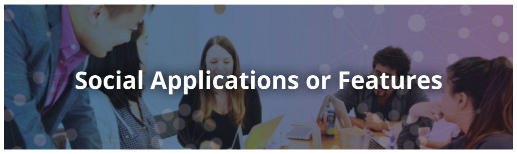
A flexible graph database model organizes and connects all of an organization's master data to provide a live, real-time 360° view of customers.

Graph technology offers an effective and efficient way to comply with sweeping regulations like the EU's General Data Protection Regulation (GDPR), which requires that businesses connect all of the data that they have about their customers and prospects. Organizations manage enterprise risk by providing both the user-facing toolkit that allows individuals to curate their own data records and the data lineage proof points to demonstrate compliance to authorities.



Graph platforms are inherently suitable for making sense of complex interdependencies central to managing networks and IT infrastructure. This is especially important in a time of increasing automation and containerization across both cloud and on-premises data centers. Graphs keep track of these interdependencies and ensure that an accurate representation of operations is available at all times, no matter how dynamic the network and IT environment.

To verify an accurate identity, the system needs to traverse through a highly interconnected dataset that is continually growing in size and complexity as employees, partners and customers enter and leave the system. Users, roles, products and permissions are not only growing in number but also in matrixed relationships where standard "tree" hierarchies are less relevant. Traditional systems no longer deliver real-time query performance required by two-factor authentication systems, resulting in long wait times for users. Using a graph database for identity and access management enables you to quickly and effectively track users, assets, devices, relationships and authorizations in this dynamic environment.

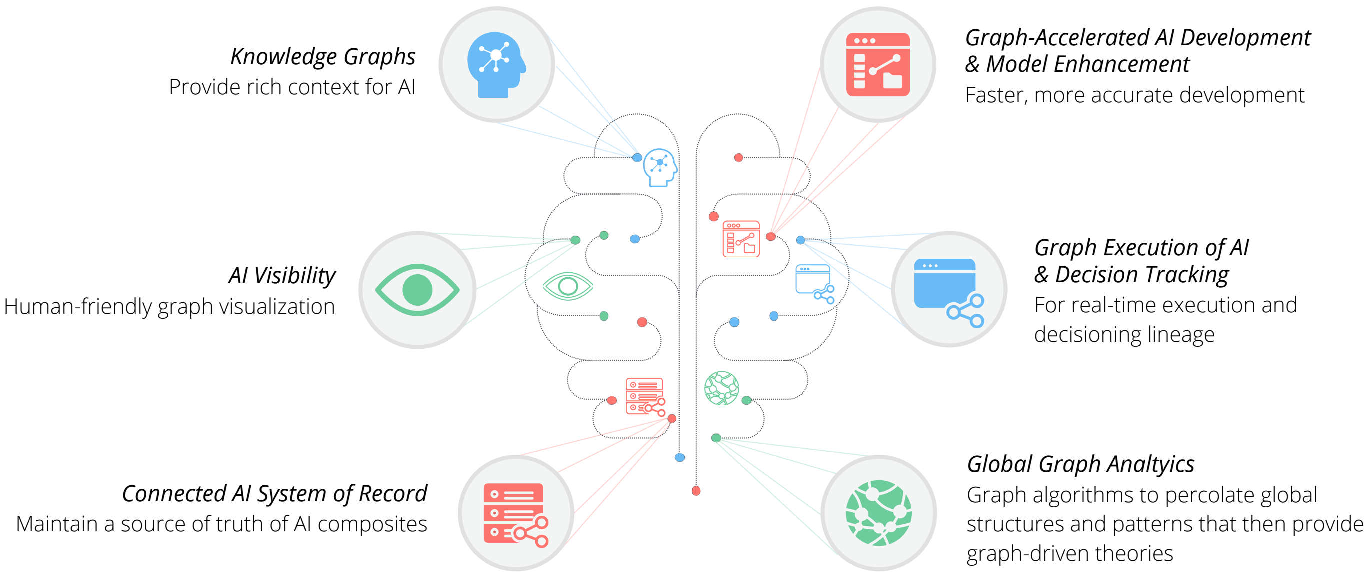


Social media networks are already graphs, so there's no point converting a graph into tables and then back again by building a social network on an RDBMS. Having a data model that directly matches your domain model helps you better understand your data, communicate more effectively and avoid needless work. A graph database such as Neo4j enables you to easily leverage social connections or infer relationships based on user activity to power your social network application or add social features to internal applications.

These are just a few of the use cases that are fueled by graph technology. Next we'll look at how graphs are supporting an emerging category of algorithms and applications based on AI.

Graph Technology & AI Applications

Graph technologies are the scaffolding for building intelligent applications, enabling more accurate predictions and faster decisions. In fact, graphs are underpinning a wide variety of artificial intelligence (AI) use cases.


Knowledge Graphs
Provide rich context for AI
Graph-Accelerated AI Development & Model Enhancement
Faster, more accurate development
AI Visibility
Human-friendly graph visualization
Graph Execution of AI & Decision Tracking
For real-time execution and decisioning lineage
Connected AI System of Record
Maintain a source of truth of AI composites
Global Graph Analytics
Graph algorithms to percolate global structures and patterns that then provide graph-driven theories

Knowledge Graphs
Andrew Ng, a preeminent thought leader in the field, includes knowledge graphs as one of the five main areas of AI. Knowledge graphs represent knowledge in a form usable by machines.

Graph analysis surfaces relationships and provides richer and deeper context for prescriptive analytics and AI applications like TextRank (a PageRank derivative) alongside natural language processing (NLP) and natural language understanding (NLU) technologies. For example, in the case of a shopping chatbot, a knowledge graph representation helps an application intelligently get from text to meaning by providing the context in which the word is used (such as the word "bat" in sports versus zoology).

Machine Learning Model Enhancement & Accelerated AI
Graphs are used to feed machine learning models and find new features to use for training, subsequently speeding up AI decisions. Graph centrality algorithms such as PageRank identify influential features to feed more accurate machine learning models and measurable predictive lift. Graph analysis computes Boolean (yes/no) answers in real time and continuously provides them as a tensor for AI recalculation and scoring.

"The major areas of artificial intelligence are speech, NLP, computer vision, machine learning, [and] knowledge graph."

– Andrew Ng

Graph Execution of AI & Decision Tracking

An operational graph – replacing a rules engine to run AI – is a natural, next step for intelligent applications. As coding AI systems in graphs becomes a norm, it will enable the tracking of AI decisions. This kind of decision tree lineage is essential for adoption and maintenance of AI logic in critical applications.

Global Graph Analytics for Theory Development

Graph analytics lift out global structures and reveal patterns in your data – without you requiring any prior knowledge of the system. For example, community detection and other algorithms are used to organize groups, suggest hierarchies, and predict missing or vulnerable relationships. In this way, you are essentially using graph-driven theory development that infers micro and macro behaviors.

AI Visibility

The adoption of AI in part depends largely on the ability to trust the results. Human-friendly graph visualizations display or explain machine learning processes that are often never exposed within ML's "black box." These visualizations serve as an abstraction to accelerate data scientists' work and to provide a visual record of how a system's logic has changed over time. Visualizations help explain and build confidence in and comfort with AI solutions.

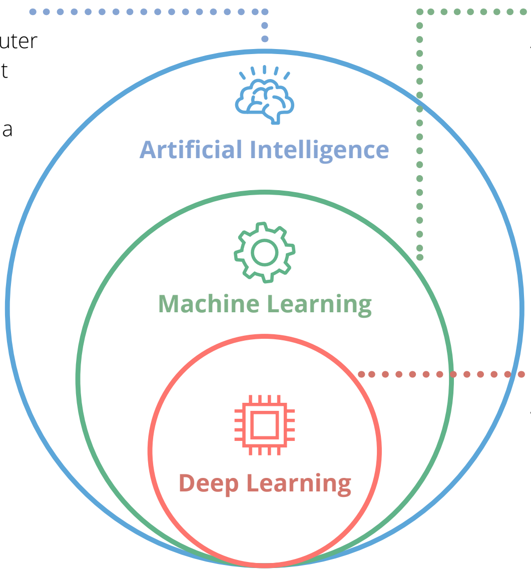
System of Record for AI Connections

Graphs serve as a source of truth for all your related AI components to create a pipeline for iterative tasks. They automate the sourcing and capture of related AI components so that data scientists focus on analysis and more easily share frameworks.

What Makes a Machine Intelligent?

While AI is the headliner, there are actually subsets of the technology that can be applied to solving human problems in different ways.

Artificial Intelligence (AI)
A process where a computer solves a task in a way that mimics human behavior. Today, narrow AI – when a machine is trained to do one particular task – is becoming more widely used, from virtual assistants to self-driving cars to automatically tagging your friends in your photos on Facebook.



Machine Learning (ML)
Algorithms that allow computers to learn from examples without being explicitly programmed.

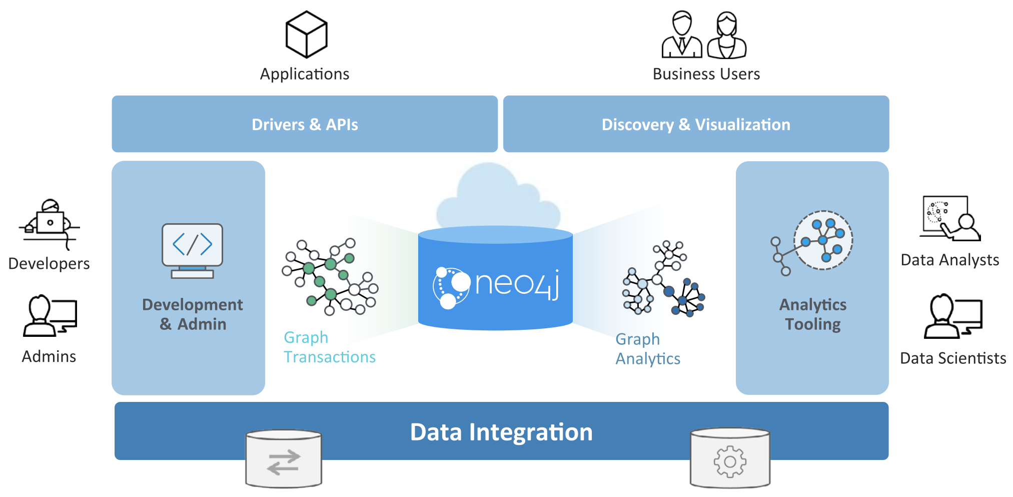
Deep Learning (DL)
A subset of ML that uses deep artificial neural networks as models and does not require feature engineering.

Source: *Curt Hopkins*

Chapter 3
Neo4j Graph Analytics

At a fundamental level, a native graph platform is required to make it easy to express relationships across many types of data elements. To succeed with connected data applications, you need to traverse these connections at speed, regardless of how many hops your query takes.

A graph platform must also offer a variety of skill-specific tools for business users, solution developers and data scientists alike. Each user group has different needs to visualize connectedness, explore query results and update information.



A graph platform like Neo4j offers an efficient means for data scientists and solutions teams to move through the stages of discovery and design.

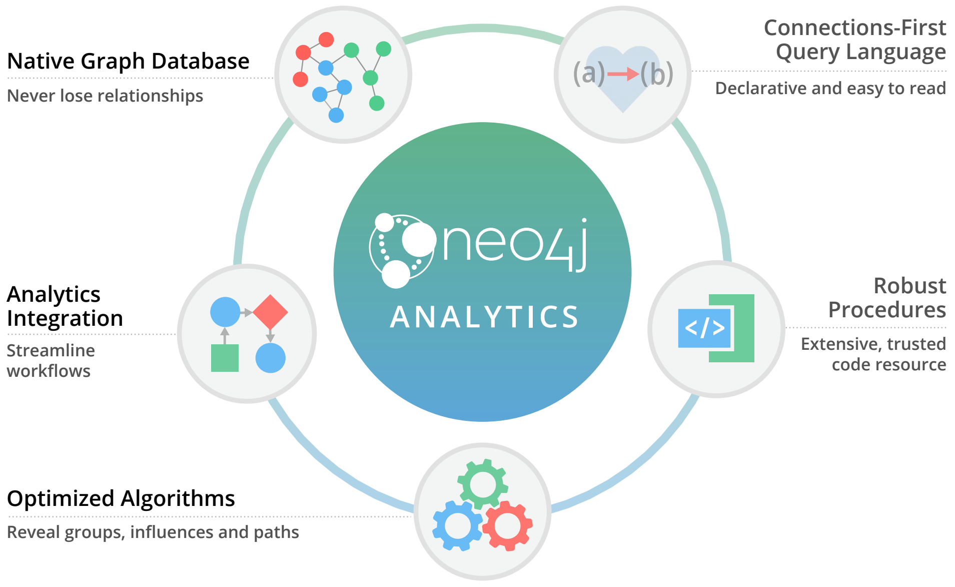
First, when exploring a concept, teams look for broad patterns and structures best served by global analysis. They need the ability to easily call upon packaged procedures and algorithms. Organizations want tools to identify communities, bottlenecks, influence points and pathways. In addition, a supported library of algorithms helps ensure that results are consistent by reducing variability introduced by many individual procedures.

In the next phase of solution modeling, a streamlined process becomes extremely important as teams must test a hypothesis and develop prototypes. And the iterative, continuous nature of the above workflow heightens the need for extremely efficient tools with fast feedback loops.

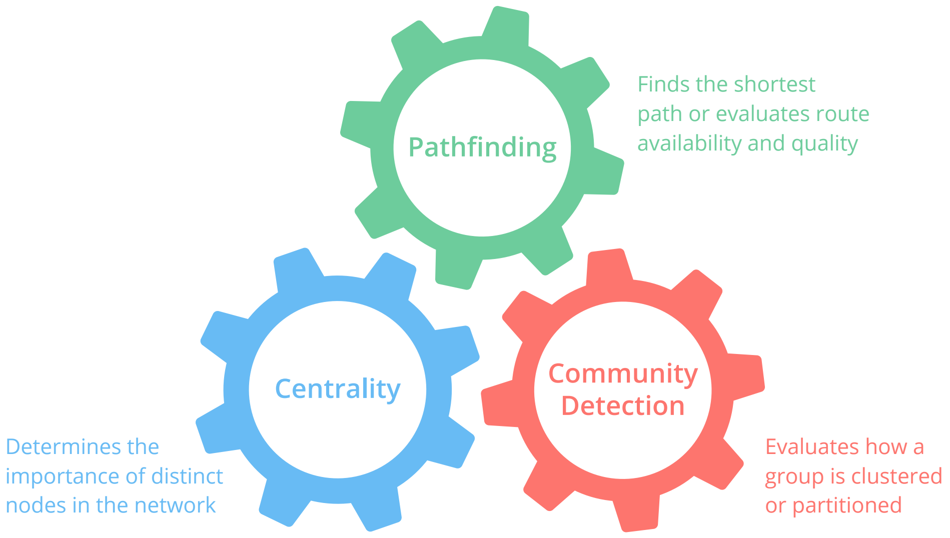
Teams will be using various data sources and tools, so a common, human-friendly way to express connections and leverage popular tools is essential.

"In fact, the rapid rise of graph technologies may signal that data connectedness is indeed a separate paradigm from the model consolidation happening across the rest of the NoSQL landscape."

– Frost & Sullivan



Neo4j offers a growing, open library of graph algorithms that are optimized for fast results. These algorithms reveal the hidden patterns and structures in your connected data around community detection, centrality and pathfinding with a core set of tested and supported algorithms.


Pathfinding
Finds the shortest path or evaluates route availability and quality
Centrality
Determines the importance of distinct nodes in the network
Community Detection
Evaluates how a group is clustered or partitioned

Graph algorithm types

Neo4j graph algorithms are simple to apply so data scientists, solution developers and operational teams can all use the same graph platform.

Neo4j graph algorithms are efficient so you analyze billions of relationships and get results in seconds to minutes, or in a few hours for more complicated queries that process large amounts of connected data.

The following table offers a sampling of problems and the specific graph algorithms that have been used to solve them.

Challenges & Graph Algorithms That Have Been Used to Solve Them[1]

Challenge	Algorithm	Algorithm Type
Figure out traffic load capacity and plan distribution or logistics in an urban area	All Pairs Shortest Path	Pathfinding
Create a low-cost tour of a travel destination	Minimum Weight Spanning Tree	Pathfinding
Identify the most influential machine learning features for extraction and model updates	PageRank	Centrality
Separate the fraudsters from the legitimate users in an online auction	Weighted Degree Centrality	Centrality
Identify the bridge points that connect separate groups	Betweenness Centrality	Centrality
Determine the delivery ETA for a package	Closeness Centrality	Centrality
Find potential duplicate records	Union Find	Community Detection
Figure out dangerous interactions between prescription drugs	Label Propagation	Community Detection
Research structures in the brain	Louvain Modularity	Community Detection

1. This list offers inspiration about the types of problems that graph algorithms have solved. Inclusion on this list does not imply that the work in question was done using Neo4j.

Part II:
Graph Algorithms in Neo4j

"Graphs are one of the unifying themes of computer science – an abstract representation that describes the organization of transportation systems, human interactions, and telecommunication networks. That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer."

– Frost & Sullivan

Chapter 4
Graph Algorithm Concepts

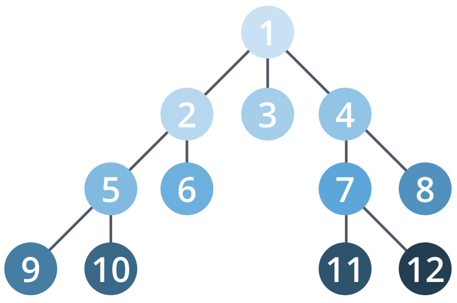
Traversal

The most fundamental graph task is to visit nodes and relationships in a methodical way; this is called traversing a graph. Traversal means moving from one item to another using predecessor and successor operations in a sorted order.

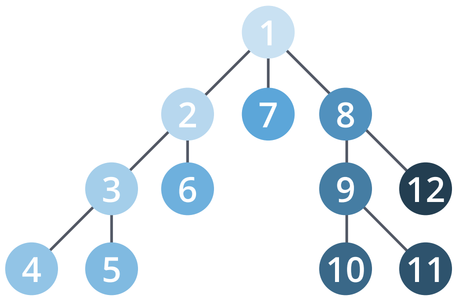
Although this sounds simple, because the sorted order is logical, the next hop is determined by a node's logical predecessor or successor and not by its physical nearness. Complexity arises as values assigned to not only nodes but relationships may be factored in. For example, in an unsorted graph, a node's predecessor would hold the largest value that is smaller than the current node's value and its successor would be the node with the smallest value that is larger.

Fundamental Traversal Algorithms

There are two fundamental graph traversal algorithms: breadth-first search (BFS) and depth-first search (DFS).



Breadth-first search



Depth-first search

The main difference between the algorithms is the order in which they explore nodes in the graph. Breadth-first search traverses a graph by exploring a node's neighbors first before considering neighbors of those neighbors, whereas depth-first search will explore as far down a path as possible, always visiting new neighbors where possible.

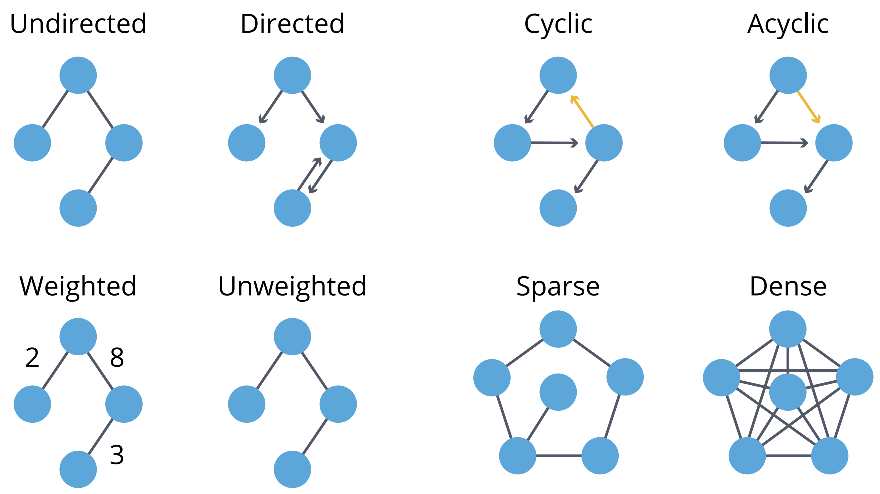
While they are not often used directly, these algorithms form an integral part of other graph algorithms:

- Depth-first search is used by the Strongly Connected Components algorithms.
- Breadth-first search is used by the Shortest Path, Closeness Centrality and Connected Components algorithms.

It's sometimes not obvious which algorithm is being used by other graph algorithms. For example, Neo4j's Shortest Path algorithm uses a fast bidirectional breadth-first search as long as any predicates can be evaluated while searching for the path.

Graph Properties

There are several basic properties of graphs that will inform your choice of how you traverse a graph and the algorithms you use.



Undirected vs. Directed. In an undirected graph, there is no direction to the relationships between nodes. For example, highways between cities are traveled in both directions. In a directed graph, relationships have one specific direction. For example, within cities, some roads are one-way streets. For some analyses, you may also want to ignore direction, for example in friendships where you want to assume the relationship is mutual. We'll also see how this is relevant to Community Detection algorithms, especially Weakly and Strongly Connected Components.

Cyclic vs. Acyclic. In graph theory, cycles are paths through relationships and nodes where you walk from and back to a particular node. There are many types of cycles within graphs, but cycles require consideration when using algorithms that may cause infinite loops, like PageRank, for example. An acyclic graph has no cycles; a tree structure is a common type of connected and acyclic (and undirected) graph.

Weighted vs. Unweighted. Weighted graphs assign values (weights) to either the nodes or their relationships; one example is the cost or time to travel a segment or the priority of a node. The shortest path through an unweighted graph is quickly found with a breadth-first search as it will always be the path with the fewest number of relationships. Weighted graphs are commonly used in pathfinding algorithms and require consideration for calculating additional values.

Sparse vs. Dense. Graphs with a large number of relationships compared to nodes are called dense. Although not strictly defined, sparse graphs are loosely linear in the number of relationships to nodes, whereas in a clearly dense graph the number of relationships would typically be the square of the nodes. Most graphs tend toward sparseness, especially where physical elements, such as pipe sizes, come into play. Care should be taken when preparing your graph data for community detection algorithms: On graphs that are extremely dense you'll find overly clustered, meaningless communities; and at the other end of the spectrum, an extremely sparse graph may find no communities at all.

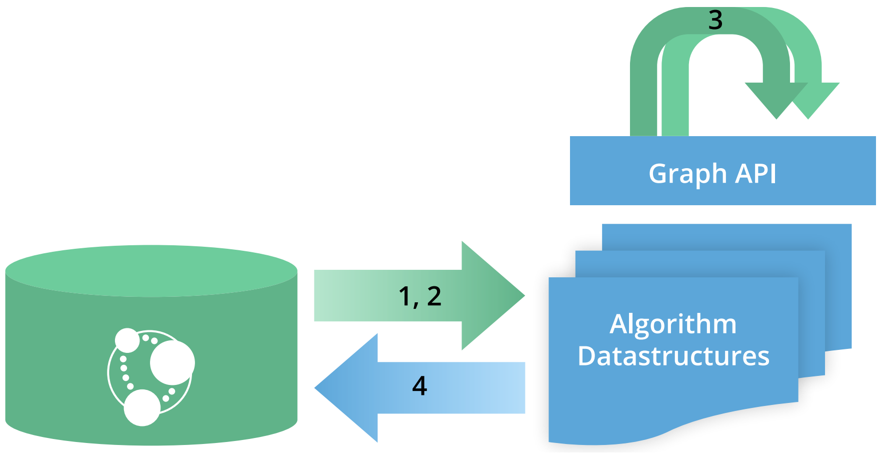
Chapter 5
The Neo4j Graph Data Science Library

The Neo4j Graph Data Science Library is used on your connected data to gain new insights more easily within Neo4j. These graph algorithms improve results from your graph data, for example by focusing on particular communities or favoring popular entities.

We developed this library as part of our effort to make it easier to use Neo4j for a wider variety of applications. These algorithms have been tuned to be as efficient as possible in regards to resource utilization as well as streamlined for management and debugging.

They are available as user-defined procedures called as part of Cypher statements running on top of Neo4j.

Here is an architecture diagram.



1. Load data in parallel from Neo4j
2. Store in efficient data structures
3. Run graph algorithm in parallel using the Graph API
4. Write data back in parallel

If you want to try out the examples in the rest of the book, you'll need to first install the graph algorithms library. Please see the "Installing Graph Algorithms" section in Appendix B.

Usage

These algorithms are exposed as Neo4j procedures. They are called directly using Cypher in your Neo4j Browser, from cypher-shell or from your client code.

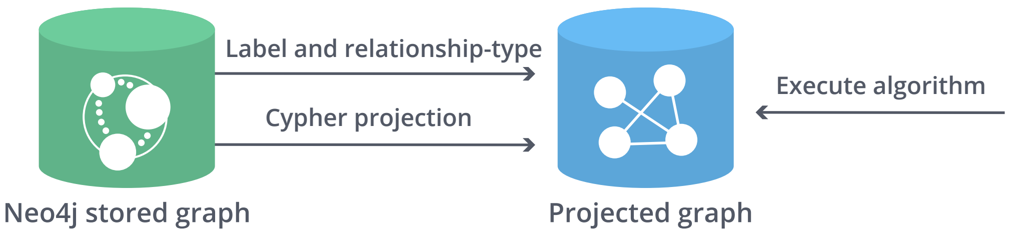
For most algorithms, there are two procedures:

- `algo.<name>` – This procedure writes results back to the graph as node-properties, and reports statistics.
- `algo.<name>.stream` – This procedure returns a stream of data. For example, node-ids and computed values.

For large graphs, the streaming procedure might return millions, or even billions, of results. In this case it may be more convenient to store the results of the algorithm, and then use them with later queries.

This is one of the use cases for a handy feature called graph projection. Graph projection places a logical subgraph into a graph algorithm when your original graph has the wrong shape or granularity for that specific algorithm. For example, if you're looking to understand the relationship between drug results for men versus women but your graph is not partitioned for this, you'll be able to temporarily project a subgraph to quickly run your algorithm upon and move on to the next step.

We project the graph we want to run algorithms on with either label and relationship-type projection, or Cypher projection.



The projected graph model is separate from Neo4j's stored graph model to enable fast caching for the topology of the graph, containing only relevant nodes, relationships and weights. During projection of a directed subgraph, only one relationship directed in and one relationship directed out is allowed between a pair of nodes. During the projection of an undirected subgraph, two relationships between a pair of nodes is allowed (there is no direction).

Label & Relationship-Type Projection

We project the subgraph we want to run the algorithm on by using the label parameter to describe nodes, and relationship-type to describe relationships.

The general call syntax is:

```
CALL algo.<name>("NodeLabel", "RelationshipType", {config})
```

Cypher Projection

If label and relationship-type projection is not selective enough to describe our subgraph to run the algorithm on, we use Cypher statements to project subsets of our graph. Use a node-statement instead of the label parameter and a relationship-statement instead of the relationship-type, and use `graph:"cypher"` in the config.

Relationships described in the relationship-statement will only be projected if both source and target nodes are described in the node-statement. Relationships that don't have both source and target nodes described in the node-statement will be ignored.

We also return a property value or weight (according to our config) in addition to the ids from these statements.

Cypher projection enables us to be more expressive in describing the subgraph that we want to analyze, but it might take longer to project the graph with more complex Cypher queries.

The general call syntax is:

```
CALL algo.<name>(
  "MATCH (n) RETURN id(n) AS id",
  "MATCH (n)-->(m) RETURN id(n) AS source, id(m) AS target",
  {graph: "cypher"})
```

Huge Graph Projection

The default label and relationship-type projection has a limitation of two billion nodes and two billion relationships, so if our projected graph is bigger than this, we need to use a huge graph projection. This is enabled by setting `graph:"huge"` in the config.

The general call syntax is:

```
CALL algo.<name>("NodeLabel", "RelationshipType", {graph: "huge"})
```

Algorithm Types

For transactions and operational decisions, you need real-time graph analysis to provide a local view of relationships between specific data items and take action. To make discoveries about the overall nature of networks and model the behavior of complex systems, you need graph algorithms that provide a broader view of patterns and structures across all data and relationships.

The following table is helpful for working out the appropriate algorithm for your use case.

Algorithm Type	Graph Problem	Examples
Pathfinding & Search	Find the optimal path or evaluate route availability and quality	• Find the quickest route to travel from A to B • Telephone call routing
Centrality	Determine the importance of distinct nodes in the networks	• Determine social media influencers • Find likely attack targets in communication and transportation networks
Community Detection	Evaluate how a group is clustered or partitioned	• Segment customers • Find potential members of a fraud ring

The next three chapters provide a reference for these three types of algorithms. They can be treated as a reference manual for the algorithms currently supported by the Neo4j Graph Platform.

If you want to try out the examples in these chapters, you'll need to install the Graph Algorithms library. Please see the "Installing Graph Algorithms" section in Appendix B.

Chapter 6
Pathfinding and Graph Search Algorithms

Pathfinding and graph search algorithms start at a node and expand relationships until the destination has been reached. Pathfinding algorithms do this while trying to find the cheapest path in terms of number of hops or weight whereas search algorithms will find a path that might not be the shortest.

Algorithm Type	What It Does	Example Uses
Shortest Path	Calculates the shortest weighted path between a pair of nodes.	Shortest Path is used for finding directions between physical locations, such as driving directions. It's also used to find the degrees of separations between people in social networks as well as their mutual connections.
Single Source Shortest Path	Calculates a path between a node and ***all*** other nodes whose summed value (weight of relationships such as cost, distance, time or capacity) to all other nodes is minimal.	Single Source Shortest Path is faster than Shortest Path and is used for the same types of problems. It's also essential in logical routing such as telephone call routing (e.g., lowest cost routing).
All Pairs Shortest Path	Calculates a shortest path forest (group) containing ***all*** shortest paths between ***all nodes*** in the graph. Commonly used for understanding alternate routing when the shortest route is blocked or becomes suboptimal.	All Pairs Shortest Path is used to evaluate alternate routes for situations such as a freeway backup or network capacity. It's also key in logical routing to offer multiple paths, for example, call routing alternatives in case of a failure.
Minimum Weight Spanning Tree	Calculates the paths along a connected tree structure with the smallest value (weight of the relationship such as cost, time or capacity) associated with visiting all nodes in the tree. It's also employed to approximate some problems with unknown compute times such as the traveling salesman problem and randomized or iterative rounding.	Minimum Weight Spanning Tree is widely used for network designs: least cost logical or physical routing such as laying cable, fastest garbage collection routes, capacity for water systems, efficient circuit designs and much more. It also has real-time applications with rolling optimizations such as processes in a chemical refinery or driving route corrections.

Shortest Path

The Shortest Path algorithm calculates the shortest (weighted) path between a pair of nodes. In this category, Dijkstra's algorithm is the most well known. It is a real-time graph algorithm, and is used as part of the normal user flow in a web or mobile application.

Pathfinding has a long history and is considered to be one of the classical graph problems; it has been researched as far back as the 19th century. It gained prominence in the early 1950s in the context of alternate routing, that is, finding the second shortest route if the shortest route is blocked.

Edsger Dijkstra came up with his algorithm in 1956 while trying to show off the new ARMAC computers. He needed to find a problem and a solution that people not familiar with computing would be able to understand, and he designed what is now known as Dijkstra's algorithm. He later implemented it for a slightly simplified transportation map of 64 cities in the Netherlands.

When Should I Use Shortest Path?

- Finding directions between physical locations. This is the most common usage, and web mapping tools such as Google Maps use the shortest path algorithm, or a variant of it, to provide driving directions.
- Social networks use the algorithm to find the degrees of separation between people. For example, when you view someone's profile on LinkedIn, it will indicate how many people separate you in the connections graph, as well as listing your mutual connections.

TIP: Dijkstra does not support negative weights. The algorithm assumes that adding a relationship to a path can never make a path shorter — an invariant that would be violated with negative weights.

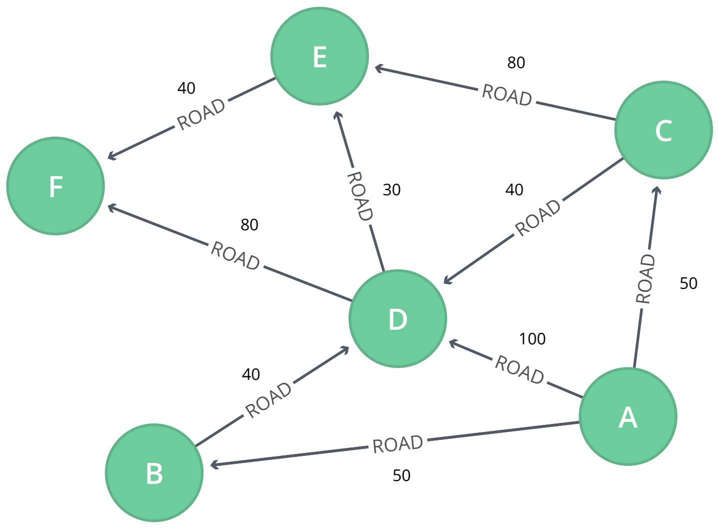
Shortest Path Example

Let's calculate Shortest Path on a small dataset.

The following Cypher statement creates a sample graph containing locations and connections between them.

```
MERGE (a:Loc {name:"A"})
MERGE (b:Loc {name:"B"})
MERGE (c:Loc {name:"C"})
MERGE (d:Loc {name:"D"})
MERGE (e:Loc {name:"E"})
MERGE (f:Loc {name:"F"})

MERGE (a)-[:ROAD {cost:50}]->(b)
MERGE (a)-[:ROAD {cost:50}]->(c)
MERGE (a)-[:ROAD {cost:100}]->(d)
MERGE (b)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:80}]->(e)
MERGE (d)-[:ROAD {cost:30}]->(e)
MERGE (d)-[:ROAD {cost:80}]->(f)
MERGE (e)-[:ROAD {cost:40}]->(f);
```



Graph Model

Now we can run the Shortest Path algorithm to find the shortest path between `A` and `F`. Execute the following query.

```
MATCH (start:Loc{name:"A"}), (end:Loc{name:"F"})
CALL algo.shortestPath.stream(start, end, "cost")
YIELD nodeId, cost
MATCH (other:Loc) WHERE id(other) = nodeId
RETURN other.name AS name, cost
```

Results

Name	Cost
A	0
C	50
D	90
E	120
F	160

The quickest route takes us from `A` to `F`, via `C`, `D`, and `E`, at a total cost of 160:

- First, we go from `A` to `C`, at a cost of 50.
- Then, we go from `C` to `D`, for an additional 40.
- Then, from `D` to `E`, for an additional 30.
- Finally, from `E` to `F`, for a further 40.

Single Source Shortest Path

The Single Source Shortest Path (SSSP) algorithm calculates the shortest (weighted) path from a node to all other nodes in the graph.

SSSP came into prominence at the same time as the Shortest Path algorithm and Dijkstra's algorithm acts as an implementation for both problems.

Neo4j implements a variation of SSSP, the delta-stepping algorithm. The delta-stepping algorithm outperforms Dijkstra's and efficiently works in sequential and parallel settings for many types of graphs.

When Should I Use Single Source Shortest Path?

Open Shortest Path First is a routing protocol for IP networks. It uses Dijkstra's algorithm to help detect changes in topology, such as link failures, and come up with a new routing structure in seconds.

TIP: Delta-stepping does not support negative weights. The algorithm assumes that adding a relationship to a path never makes a path shorter – an invariant that would be violated with negative weights.

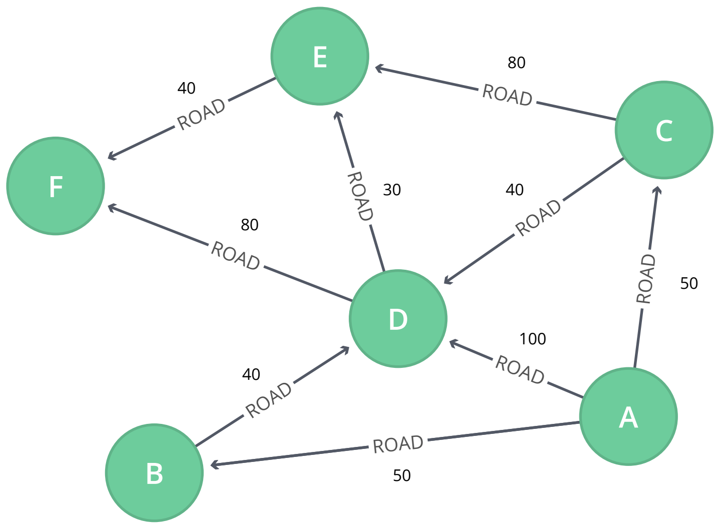
Single Source Shortest Path Example

Let's calculate Single Source Shortest Path on a small dataset.

The following Cypher statement creates a sample graph containing locations and connections between them.

```
MERGE (a:Loc {name:"A"})
MERGE (b:Loc {name:"B"})
MERGE (c:Loc {name:"C"})
MERGE (d:Loc {name:"D"})
MERGE (e:Loc {name:"E"})
MERGE (f:Loc {name:"F"})

MERGE (a)-[:ROAD {cost:50}]->(b)
MERGE (a)-[:ROAD {cost:50}]->(c)
MERGE (a)-[:ROAD {cost:100}]->(d)
MERGE (b)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:80}]->(e)
MERGE (d)-[:ROAD {cost:30}]->(e)
MERGE (d)-[:ROAD {cost:80}]->(f)
MERGE (e)-[:ROAD {cost:40}]->(f);
```



Graph Model

Now we can run the Single Source Shortest Path algorithm to find the shortest path between A and all other nodes. Execute the following query.

```
MATCH (n:Loc {name:"A"})
CALL algo.shortestPath.deltaStepping.stream(n, "cost", 3.0
YIELD nodeId, distance

MATCH (destination) WHERE id(destination) = nodeId

RETURN destination.name AS destination, distance
```

Results

Name	Cost
A	0
B	50
C	50
D	90
E	120
F	160

The above table shows the cost of going from A to each of the other nodes, including itself at a cost of 0.

All Pairs Shortest Path

The All Pairs Shortest Path (APSP) algorithm calculates the shortest (weighted) path between all pairs of nodes. This algorithm has optimizations that make it quicker than calling the Single Source Shortest Path algorithm for every pair of nodes in the graph.

Some pairs of nodes might not be reachable from each other, so no shortest path exists between these pairs. In this scenario, the algorithm returns infinity value as a result between these pairs of nodes.

When Should I Use All Pairs Shortest Path?

- The All Pairs Shortest Path algorithm is used in urban service system problems, such as the location of urban facilities or the distribution or delivery of goods. One example of this is determining the traffic load expected on different segments of a transportation grid. For more information, see Urban Operations Research.
- All Pairs Shortest Path is used as part of the REWIRE data center design algorithm, which finds a network with maximum bandwidth and minimal latency. There are more details about this approach in the following academic paper: "REWIRE: An Optimization-based Framework for Data Center Network Design."

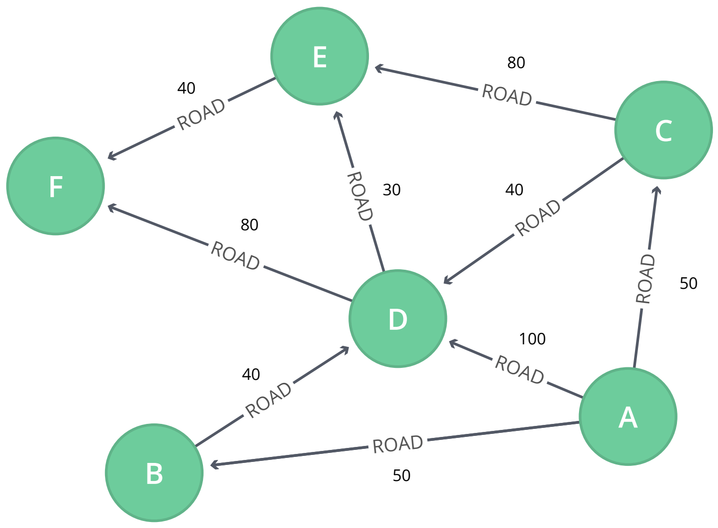
All Pairs Shortest Path Example

Let's calculate All Pairs Shortest Path on a small dataset.

The following Cypher statement creates a sample graph containing locations and connections between them.

```
MERGE (a:Loc {name:"A"})
MERGE (b:Loc {name:"B"})
MERGE (c:Loc {name:"C"})
MERGE (d:Loc {name:"D"})
MERGE (e:Loc {name:"E"})
MERGE (f:Loc {name:"F"})

MERGE (a)-[:ROAD {cost:50}]->(b)
MERGE (a)-[:ROAD {cost:50}]->(c)
MERGE (a)-[:ROAD {cost:100}]->(d)
MERGE (b)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:40}]->(d)
MERGE (c)-[:ROAD {cost:80}]->(e)
MERGE (d)-[:ROAD {cost:30}]->(e)
MERGE (d)-[:ROAD {cost:80}]->(f)
MERGE (e)-[:ROAD {cost:40}]->(f);
```



Graph Model

Now we run the All Pairs Shortest Path algorithm to find the shortest path between every pair of nodes. Execute the following query.

```
CALL algo.allShortestPaths.stream("cost",{nodeQuery:"Loc",defaultValue:1.0})
YIELD sourceNodeId, targetNodeId, distance
WITH sourceNodeId, targetNodeId, distance
WHERE algo.isFinite(distance) = true

MATCH (source:Loc) WHERE id(source) = sourceNodeId
MATCH (target:Loc) WHERE id(target) = targetNodeId
WITH source, target, distance WHERE source <> target

RETURN source.name AS source, target.name AS target, distance
ORDER BY distance DESC
LIMIT 10
```

Results

Name	Target	Cost
A	F	100
C	F	90
B	F	90
A	E	80
C	E	70
B	E	80
A	B	50
D	F	50
A	C	50
A	D	50

This query returned the top 10 pairs of nodes that are the furthest away from each other. `F` and `E` appear to be the most distant from the others.

Minimum Weight Spanning Tree

The Minimum Weight Spanning Tree starts from a given node, and finds all its reachable nodes and the set of relationships that connect the nodes together with the minimum possible weight. Prim's algorithm is one of the simplest and best-known Minimum Weight Spanning Tree algorithms. The K-Means variant of this algorithm can be used to detect clusters in the graph.

The first known algorithm for finding a minimum weight spanning tree was developed by the Czech scientist Otakar Borůvka in 1926 while trying to design an efficient electricity network for Moravia. Prim's algorithm was invented by Jarnik in 1930 and rediscovered by Prim in 1957. It is similar to Dijkstra's Shortest Path algorithm, but rather than minimizing the total length of a path ending at each relationship, it minimizes the length of each relationship individually. Unlike Dijkstra's, Prim's tolerates negative-weight relationships.

NOTE: The algorithm operates as follows:

- Start with a tree containing only one node (and no relationships).
- Select the minimal-weight relationship coming from that node and add it to our tree.
- Repeatedly choose a minimal-weight relationship that joins any node in the tree to one that is not in the tree, adding the new relationship and node to our tree.
- When there are no more nodes to add, the tree we have built is a minimum spanning tree.

When Should I Use Minimum Weight Spanning Tree?

- Minimum Weight Spanning Tree was applied to analyze airline and sea connections of Papua New Guinea and minimize the travel cost of exploring the country. It could be used to help design low-cost tours that visit many destinations across a country. The research mentioned is found here: "An Application of Minimum Spanning Trees to Travel Planning."
- Minimum Weight Spanning Tree has been used to analyze and visualize correlations in a network of currencies based on the correlation between currency returns. This is described in "Minimum Spanning Tree Application in the Currency Market."
- Exhaustive clinical research has shown Minimum Weight Spanning Tree to be useful in tracing the history of infection transmission in an outbreak. For more information, see "Use of the Minimum Spanning Tree Model for Molecular Epidemiological Investigation of a Nosocomial Outbreak of Hepatitis C Virus Infection."

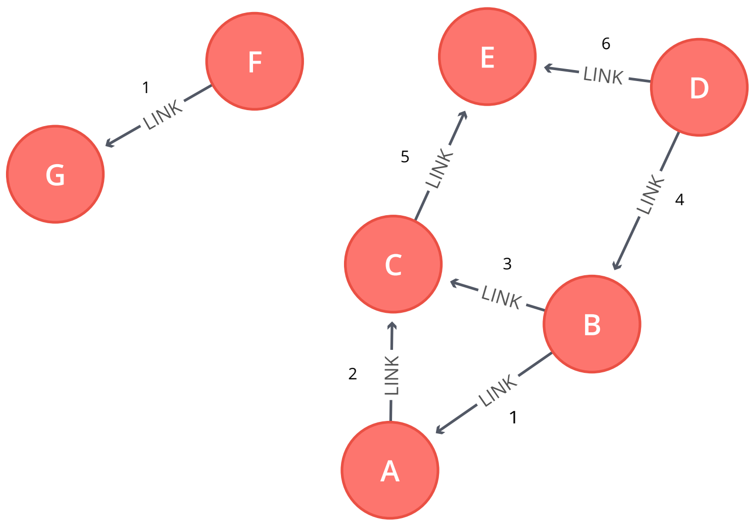
TIP: The Minimum Weight Spanning Tree algorithm only gives meaningful results when run on a graph where the relationships have different weights. If the graph has no weights, or all relationships have the same weight, then any spanning tree is a minimum spanning tree.

Minimum Weight Spanning Tree Example

Let's see the Minimum Weight Spanning Tree algorithm in action. The following Cypher statement creates a graph containing places and links between them.

```
MERGE (a:Place {id:"A"})
MERGE (b:Place {id:"B"})
MERGE (c:Place {id:"C"})
MERGE (d:Place {id:"D"})
MERGE (e:Place {id:"E"})
MERGE (f:Place {id:"F"})
MERGE (g:Place {id:"G"})

MERGE (d)-[:LINK {cost:4}]->(b)
MERGE (d)-[:LINK {cost:6}]->(e)
MERGE (b)-[:LINK {cost:1}]->(a)
MERGE (b)-[:LINK {cost:3}]->(c)
MERGE (a)-[:LINK {cost:2}]->(c)
MERGE (c)-[:LINK {cost:5}]->(e)
MERGE (f)-[:LINK {cost:1}]->(g);
```



Graph Model

We run the algorithm to find the Minimum Weight Spanning Tree starting from `D` by executing the following query.

```
MATCH (n:Place {id:"D"})
CALL algo.spanningTree.minimum("Place", "LINK", "cost", id(n),
  {write:true, writeProperty:"MINST"})
YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN loadMillis, computeMillis, writeMillis, effectiveNodeCount;
```

This procedure creates `MINST` relationships representing the minimum spanning tree. We then run the following query to find all pairs of nodes and the associated cost of the relationships between them.

```
MATCH path = (n:Place {id:"D"})-[:MINST*]-()
WITH relationships(path) AS rels
UNWIND rels AS rel
WITH DISTINCT rel AS rel
RETURN startNode(rel).id AS source, endNode(rel).id AS destination, rel.cost AS cost
```

Results

Source	Destination	Cost
D	B	4
B	A	1
A	C	2
C	E	5

The Minimum Weight Spanning Tree excludes the relationship with cost 6 from `D` to `E`, and the one with cost 3 from `B` to `C`. Nodes `F` and `G` aren't included because they're unreachable from `D`.

There are also variations of the algorithm that find the maximum weight spanning tree or k-spanning tree.

Chapter 7
Centrality Algorithms

Centrality algorithms are used to find the most influential nodes in a graph. Many of these algorithms were invented in the field of social network analysis.

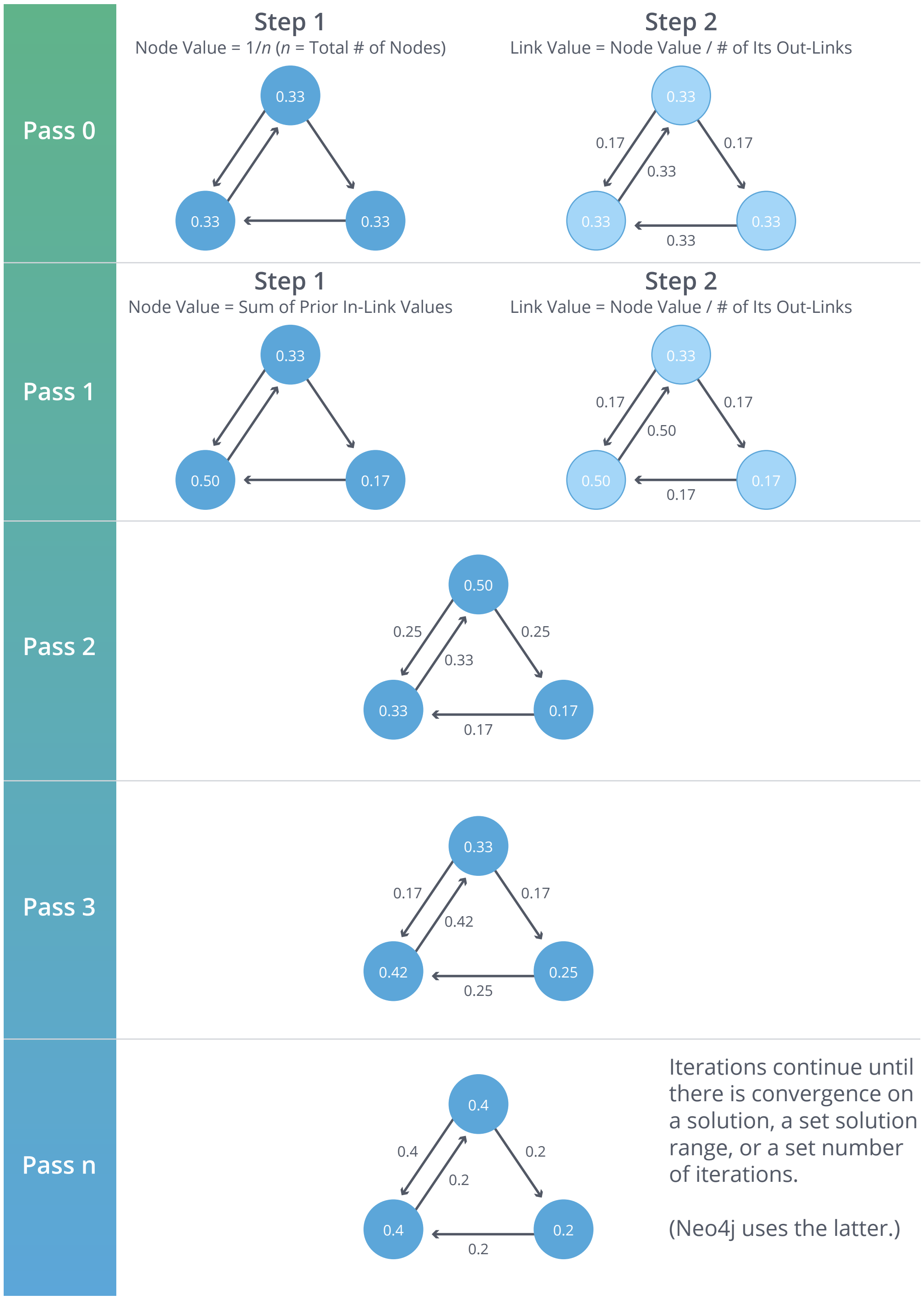
Algorithm Type	What It Does	Example Uses
PageRank	Estimates a current node's importance from its linked neighbors and then again from their neighbors. A node's rank is derived from the number and quality of its transitive links to estimate influence. Although popularized by Google, it's widely recognized as a way of detecting influential nodes in any network.	PageRank is used to estimate importance and influence. It's used to suggest Twitter accounts to follow and for general sentiment analysis. PageRank is also used in machine learning to identify the most influential features for extraction as well as ranking text for entity relevance in natural language processing. In biology, it's been used to identify which species extinctions within a food web would lead to the biggest chain-reaction of species death.
Degree Centrality	Measures the number of relationships a node has. It's broken into indegree (flowing in) and outdegree (flowing out) where relationships are directed.	Degree Centrality looks at immediate connectedness for uses such as evaluating the near-term risk of a person catching a virus or the probability of a person hearing a given piece of information. In social studies, indegree of a node is used to estimate popularity and outdegree of a node is used for gregariousness..
Betweenness Centrality	Measures the number of shortest paths that pass through a node. Nodes that most frequently lie on shortest paths have higher betweenness centrality scores and are the bridges between different clusters. It is often associated with the control over the flow of resources and information.	Betweenness Centrality applies to a wide range of problems in network science and it pinpoints bottlenecks or vulnerabilities in communication and transportation networks. In genomics, it helps researchers understand the control certain genes have in protein networks for improvements such as better drug disease targeting. Betweenness Centrality has also been used to evaluate information flows among multiplayer online gamers in addition to analyzing expertise sharing in communities of physicians.
Closeness Centrality	Measures how central a node is within its cluster. Nodes with the shortest paths to all other nodes are assumed to be able to reach the entire group the fastest.	Closeness Centrality is applicable in a number of resource, communication and behavioral analyses, especially when interaction speed is significant. It has been used in identifying the best location of new public services for maximum accessibility. In social analysis, it helps find people with the ideal social network location for faster dissemination of information.

TIP: Several of the centrality algorithms calculate shortest paths between every pair of nodes and can therefore run for a long time. This works well for small- to medium-sized graphs but can be prohibitive for large graphs. Some algorithms (for example, Betweenness Centrality) have approximating versions that are used to address longer runtimes or larger graphs.

PageRank

PageRank is an algorithm that measures the transitive, or directional, influence of nodes. All other centrality algorithms we discuss measure the direct influence of a node, whereas PageRank considers the influence of your neighbors and ***their*** neighbors. For example, having a few influential friends could raise your PageRank more than just having a lot of low-influence friends.

PageRank is computed by either iteratively distributing one node's rank (originally based on degree) over its neighbors or by randomly traversing the graph and counting the frequency of hitting each node during these walks.



Iterations continue until there is convergence on a solution, a set solution range, or a set number of iterations.

(Neo4j uses the latter.)

PageRank is named after Google co-founder Larry Page and is used to rank websites in Google's search results. It counts the number, and quality, of links to a page, which determines an estimation of how important the page is. The underlying assumption is that pages of importance are more likely to receive a higher volume of links from other influential pages.

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

```
PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))
```

where,

- we assume that a page `A` has pages `T1` to `Tn` which point to it (i.e., are citations).
- `d` is a damping factor which is set between 0 and 1. It is usually set to 0.85.
- `C(A)` is defined as the number of links going out of page `A`.

When Should I Use PageRank?

PageRank can be applied across a wide range of domains. The following are some notable use cases:

- Personalized PageRank is used by Twitter to present users with recommendations of other accounts that they may wish to follow. The algorithm is run over a graph that contains shared interests and common connections. Their approach is described in more detail in "WTF: The Who to Follow Service at Twitter."
- PageRank has been used to rank public spaces or streets, predicting traffic flow and human movement in these areas. The algorithm is run over a graph that contains intersections connected by roads, where the PageRank score reflects the tendency of people to park, or end their journey, on each street. This is described in more detail in "Self-organized Natural Roads for Predicting Traffic Flow: A Sensitivity Study."
- PageRank is also used as part of an anomaly or fraud detection system in the healthcare and insurance industries. It helps find doctors or providers that are behaving in an unusual manner and then feeds the score into a machine learning algorithm.

There are many more use cases for PageRank described in David Gleich's paper, "PageRank Beyond the Web."

TIP: There are some things to be aware of when using the PageRank algorithm:

- If there are no links from within a group of pages to outside of the group, then the group is considered a spider trap.
- Rank sink occurs when a network of pages forms an infinite cycle.
- Dead-ends occur when pages have no out-links. If a page contains a link to a dead-end page, the link is known as a dangling link.

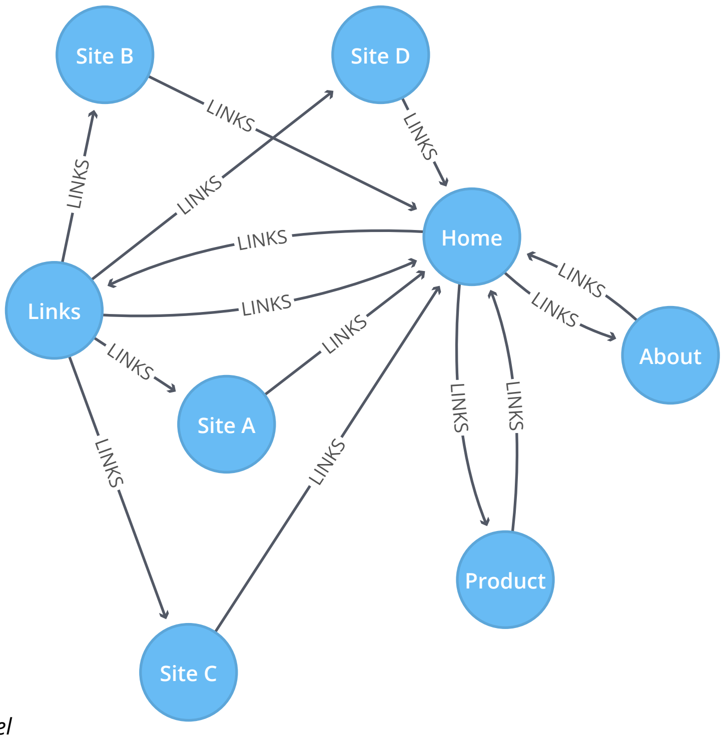
If you see unexpected results from running the algorithm, it is worth doing some exploratory analysis of the graph to see if any of these problems are the cause. You can read The Google PageRank Algorithm and How It Works to learn more.

PageRank Example

Let's calculate PageRank on a small dataset. The following Cypher statement creates a sample graph of web pages and links between them.

```
MERGE (home:Page {name:"Home"})
MERGE (about:Page {name:"About"})
MERGE (product:Page {name:"Product"})
MERGE (links:Page {name:"Links"})
MERGE (a:Page {name:"Site A"})
MERGE (b:Page {name:"Site B"})
MERGE (c:Page {name:"Site C"})
MERGE (d:Page {name:"Site D"})

MERGE (home)-[:LINKS]->(about)
MERGE (about)-[:LINKS]->(home)
MERGE (product)-[:LINKS]->(home)
MERGE (home)-[:LINKS]->(product)
MERGE (links)-[:LINKS]->(home)
MERGE (home)-[:LINKS]->(links)
MERGE (links)-[:LINKS]->(a)
MERGE (a)-[:LINKS]->(home)
MERGE (links)-[:LINKS]->(b)
MERGE (b)-[:LINKS]->(home)
MERGE (links)-[:LINKS]->(c)
MERGE (c)-[:LINKS]->(home)
MERGE (links)-[:LINKS]->(d)
MERGE (d)-[:LINKS]->(home
```



Graph Model

Now we run the PageRank algorithm to calculate the most influential pages. Execute the following query.

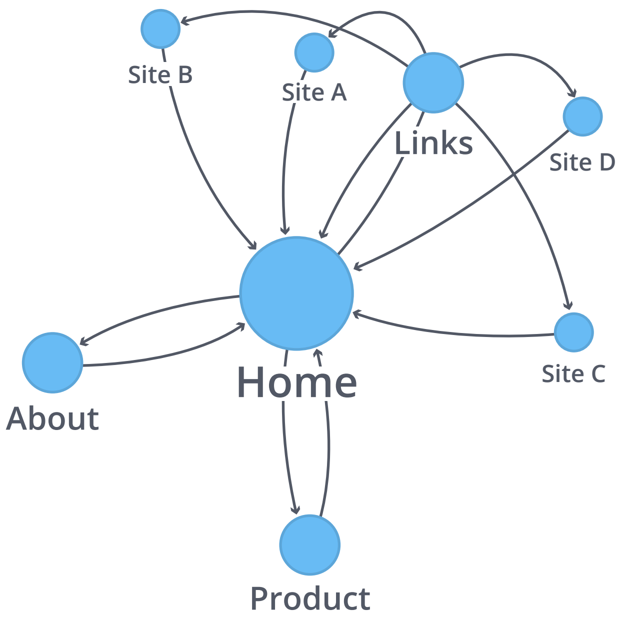
```
CALL algo.pageRank.stream("Page", "LINKS",
{iterations:20})
YIELD nodeId, score

MATCH (node) WHERE id(node) = nodeId

RETURN node.name AS page,score
ORDER BY score DESC
```

Results

Name	PageRank
Home	3.232
Product	1.059
Links	1.059
About	1.059
Site A	0.328
Site B	0.328
Site C	0.328
Site D	0.328



Visualization of PageRank

As we might expect, the Home page has the highest PageRank because it has incoming links from all other pages. Also, it's not only the number of incoming links that is important, but also the importance of the pages behind those links.

Degree Centrality

Degree Centrality is the simplest of all the centrality algorithms. It measures the number of incoming and outgoing relationships from a node.

The algorithm helps us find popular nodes in a graph.

Degree Centrality was proposed by Linton C. Freeman in his 1979 paper, "Centrality in Social Networks Conceptual Clarification." While the algorithm is usually used to find the popularity of individual nodes, it is often used as part of a global analysis where we calculate the minimum degree, maximum degree, mean degree, and standard deviation across the whole graph.

When Should I Use Degree Centrality?

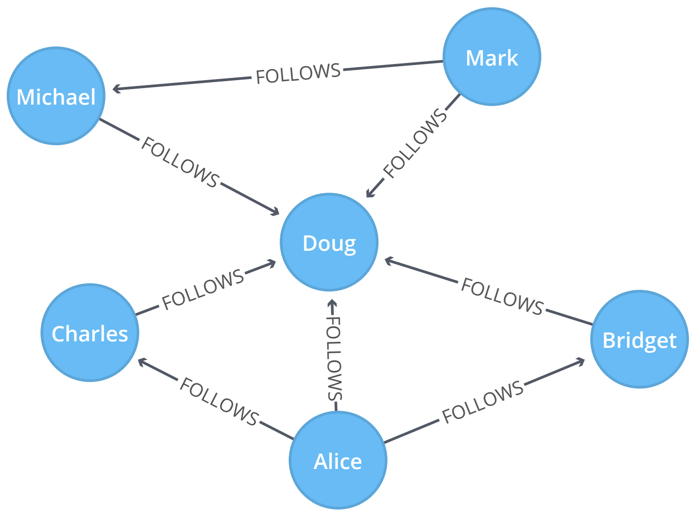
- Degree Centrality is an important component of any attempt to analyze influence by looking at the number of incoming and outgoing relationships, such as connections of people on a social network. For example, in BrandWatch's most influential men and women on Twitter 2017, the top five people in each category have over 40 million followers each.
- Weighted Degree Centrality has been used to help separate fraudsters from legitimate users of an online auction. The weighted centrality for fraudsters is significantly higher because they tend to collude with each other to artificially increase the price of items. Read more in "Two Step graph-based semi-supervised Learning for Online Auction Fraud Detection."

Degree Centrality Example

Let's see how Degree Centrality works on a small dataset. The following Cypher statement creates a Twitter-esque graph of users and followers.

```
MERGE (nAlice:User {id:"Alice"})
MERGE (nBridget:User {id:"Bridget"})
MERGE (nCharles:User {id:"Charles"})
MERGE (nDoug:User {id:"Doug"})
MERGE (nMark:User {id:"Mark"})
MERGE (nMichael:User {id:"Michael"})

MERGE (nAlice)-[:FOLLOWS]->(nDoug)
MERGE (nAlice)-[:FOLLOWS]->(nBridget)
MERGE (nAlice)-[:FOLLOWS]->(nCharles)
MERGE (nMark)-[:FOLLOWS]->(nDoug)
MERGE (nMark)-[:FOLLOWS]->(nMichael)
MERGE (nBridget)-[:FOLLOWS]->(nDoug)
MERGE (nCharles)-[:FOLLOWS]->(nDoug)
MERGE (nMichael)-[:FOLLOWS]->(nDoug)
```



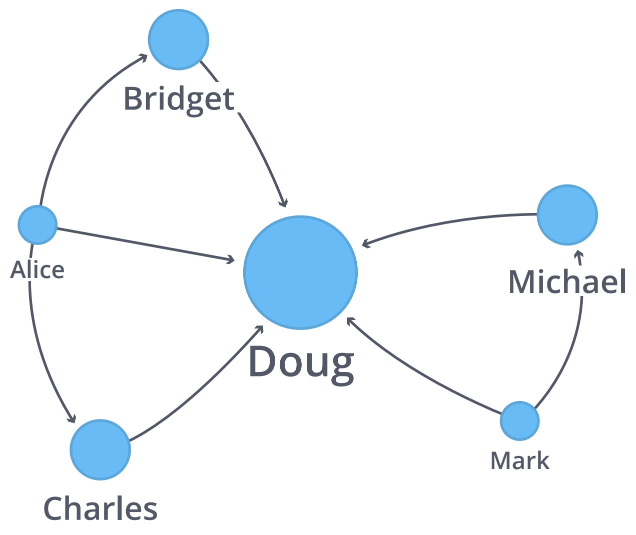
Graph Model

The following query calculates the number of people that each user follows and is followed by.

```
MATCH (u:User)
RETURN u.id AS name,
       size((u)-[:FOLLOWS]->()) AS follows,
       size((u)<-[:FOLLOWS]-()) AS followers
```

Results

Name	Following	Followers
Alice	3	0
Bridget	1	1
Charles	1	1
Doug	0	5
Mark	2	0
Michael	1	1



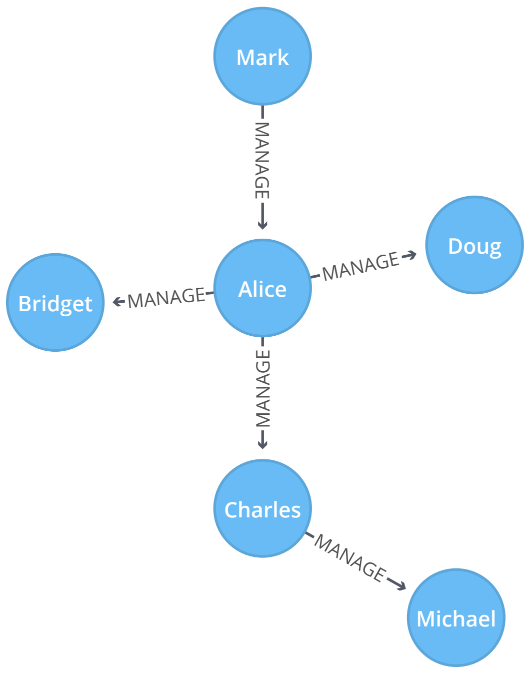
Visualization of Degree Centrality

Doug is the most popular user in our imaginary Twitter graph with five followers; all other users follow him but he doesn't follow anybody back. In the real Twitter network, celebrities have high follower counts but tend to follow few people. We could therefore consider Doug a celebrity!

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.

In the following example, Alice is the main connection in the graph.



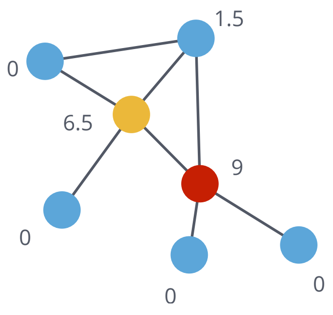
Alice

If Alice is removed, all connections in the graph would be cut off. This makes Alice important, because she ensures that no nodes are isolated.

The Betweenness Centrality algorithm calculates the shortest (weighted) path between every pair of nodes in a connected graph, using the breadth-first search algorithm. Each node receives a score, based on the number of these shortest paths that pass through the node. Nodes that most frequently lie on these shortest paths will have a higher betweenness centrality score.

The algorithm was given its first formal definition by Linton Freeman in his 1971 paper, "A Set of Measures of Centrality Based on Betweenness." It was considered to be one of the three distinct intuitive conceptions of centrality.

The algorithm operates as follows:



- First, find all shortest paths
- Then, for each node, divide the number of shortest paths that go through that node by the total number of shortest paths in the graph
- The higher scores, red node and then yellow node, have the highest betweenness centrality

Betweenness Centrality

When Should I Use Betweenness Centrality?

- Betweenness Centrality is used to research the network flow in a package delivery process or in a telecommunications network. These networks are characterized by traffic that has a known target and takes the shortest path possible. This, and other scenarios, are described by Stephen P. Borgatti in "Centrality and network flow."
- Betweenness Centrality is used to identify influencers in legitimate or criminal organizations. Studies show that influencers in organizations are not necessarily in management positions, but instead are found in brokerage positions of the organizational network. Removal of such influencers could seriously destabilize the organization. More details are found in "Brokerage qualifications in ringing operations" by Carlo Morselli and Julie Roy.
- Betweenness Centrality is also used to help microbloggers spread their reach on Twitter, with a recommendation engine that targets influencers that they should interact with in the future. This approach is described in "Making Recommendations in a Microblog to Improve the Impact of a Focal User."

TIP:

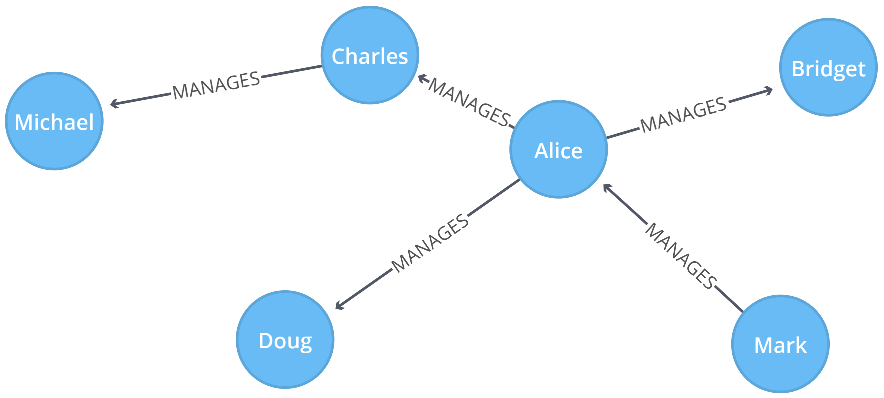
- Betweenness Centrality makes the assumption that all communication between nodes happens along the shortest path and with the same frequency, which isn't always the case in real life. Therefore, it doesn't give us a perfect view of the most influential nodes in a graph, but rather a good representation. Newman explains this in more detail on page 186 of Networks: An Introduction.
- For large graphs, exact centrality computation isn't practical. The fastest known algorithm for exactly computing betweenness of all the nodes requires at least *O(nm)* time for unweighted graphs, where *n* is the number of nodes and *m* is the number of relationships. Instead, we use an approximation algorithm that works with a subset of nodes.

Betweenness Centrality Example

Let's see how Betweenness Centrality works on a small dataset. The following Cypher statement creates an organizational hierarchy.

```
MERGE (nAlice:User {id:"Alice"})
MERGE (nBridget:User {id:"Bridget"})
MERGE (nCharles:User {id:"Charles"})
MERGE (nDoug:User {id:"Doug"})
MERGE (nMark:User {id:"Mark"})
MERGE (nMichael:User {id:"Michael"})

MERGE (nAlice)-[:MANAGES]->(nBridget)
MERGE (nAlice)-[:MANAGES]->(nCharles)
MERGE (nAlice)-[:MANAGES]->(nDoug)
MERGE (nMark)-[:MANAGES]->(nAlice)
MERGE (nCharles)-[:MANAGES]->(nMichael);
```



Graph Model

The following query executes the Betweenness Centrality algorithm.

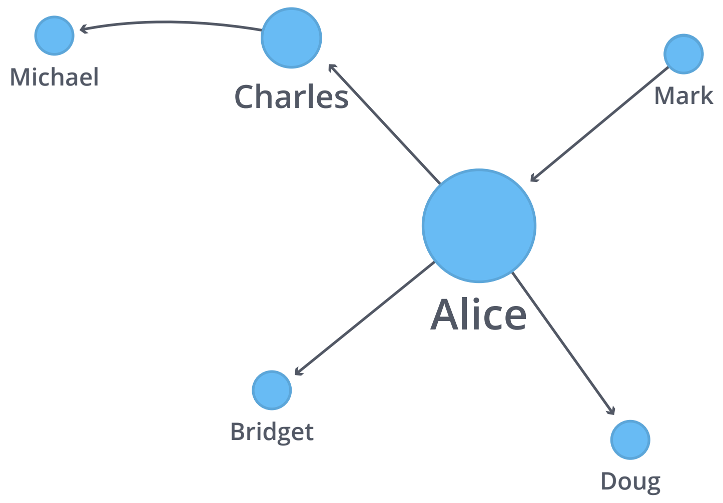
```
CALL algo.betweenness.stream("User", "MANAGES", {direction:"out"})
YIELD nodeId, centrality

MATCH (user:User) WHERE id(user) = nodeId

RETURN user.id AS user,centrality
ORDER BY centrality DESC;
```

Results

Name	Centrality Weight
Alice	4
Charles	2
Bridget	0
Michael	0
Doug	0
Mark	0



Visualization of Betweenness Centrality

Alice is the main broker in this network, and Charles is a minor broker. The others don't have any influence, because all the shortest paths between pairs of people go via Alice or Charles.

Approximation of Betweenness Centrality

As mentioned above, calculating the exact betweenness centrality on large graphs can be very time consuming. Therefore, you might choose to use an approximation algorithm that runs much quicker and still provides useful information.

The RA-Brandes algorithm is the best-known algorithm for calculating an approximate score for betweenness centrality. Rather than calculating the shortest path between every pair of nodes, the RA-Brandes algorithm considers only a subset of nodes. Two common strategies for selecting the subset of nodes are:

Random
Nodes are selected uniformly, at random, with defined probability of selection. The default probability is *log10(N) / e^2*. If the probability is 1, then the algorithm works the same way as the normal Betweenness Centrality algorithm, where all nodes are loaded.

Degree
First, the mean degree of the nodes is calculated, and then only the nodes whose degree is higher than the mean are visited (i.e., only dense nodes are visited).

As a further optimization, you limit the depth used by the Shortest Path algorithm.

Approximation of Betweenness Centrality Example

Let's see how Approximation of Betweenness Centrality works on the same dataset that we used for the Betweenness Centrality algorithm.

The following query executes the Approximation of Betweenness Centrality algorithm.

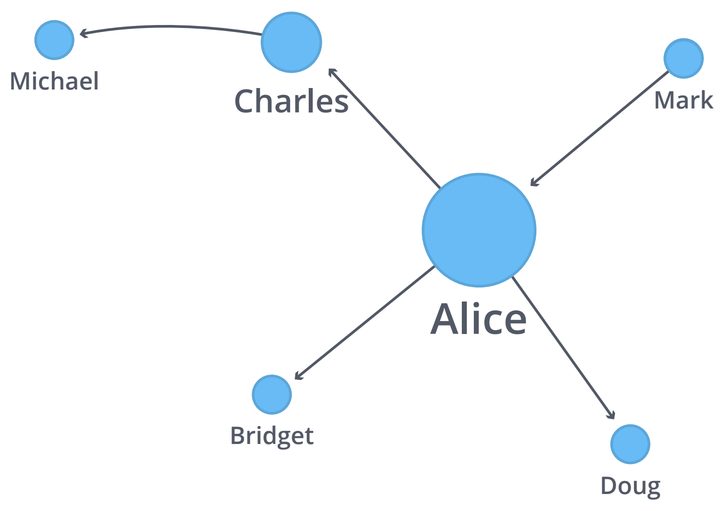
```
CALL algo.betweenness.sampled.stream("User", "MANAGES",
  {strategy:"random", probability:1.0, maxDepth:1, direction: "out"})

YIELD nodeId, centrality

MATCH (user) WHERE id(user) = nodeId
RETURN user.id AS user,centrality
ORDER BY centrality DESC;
```

Results

Name	Centrality Weight
Alice	3
Charles	1
Bridget	0
Michael	0
Doug	0
Mark	0



Visualization of Approximation of Betweenness Centrality

Alice is still the main broker in the network, and Charles is a minor broker, although their centrality score has dropped as the algorithm only considers relationships at a depth of 1. The others don't have any influence, because all the shortest paths between pairs of people go via Alice or Charles.

Closeness Centrality

Closeness Centrality is a way of detecting nodes that are able to spread information efficiently through a graph.

The closeness centrality of a node measures its average farness (inverse distance) to all other nodes. Nodes with a high closeness score have the shortest distances to all other nodes.

For each node, the Closeness Centrality algorithm calculates the sum of its distances to all other nodes, based on calculating the shortest paths between all pairs of nodes. The resulting sum is then inverted to determine the closeness centrality score for that node.

NOTE: The **raw closeness centrality** of a node is calculated using the formula:

```
raw closeness centrality(node) = 1 / sum(distance from node to
all other nodes)
```

It is more common to normalize this score so that it represents the average length of the shortest paths rather than their sum. This adjustment allows comparisons of the closeness centrality of nodes of graphs of different sizes.

The formula for **normalized closeness centrality** is as follows:

```
normalized closeness centrality(node) = (number of nodes - 1) /
sum(distance from node to all other node
```

When Should I Use Closeness Centrality?

- Closeness centrality is used to research organizational networks where individuals with high closeness centrality are in a favorable position to control and acquire vital information and resources within the organization. One such study is "Mapping Networks of Terrorist Cells" by Valdis E. Krebs.
- Closeness centrality is also interpreted as an estimated time of arrival through telecommunications or package delivery networks where content flows through shortest paths to a predefined target. It is also used in networks where information spreads through all shortest paths simultaneously, such as infections spreading through a local community. Find more details in "Centrality and Network Flow" by Stephen P. Borgatti.
- Closeness centrality helps estimate the importance of words in a document, based on a graph-based keyphrase extraction process. This process is described by Florian Boudin in "A Comparison of Centrality Measures for Graph-Based Keyphrase Extraction."

NOTE:

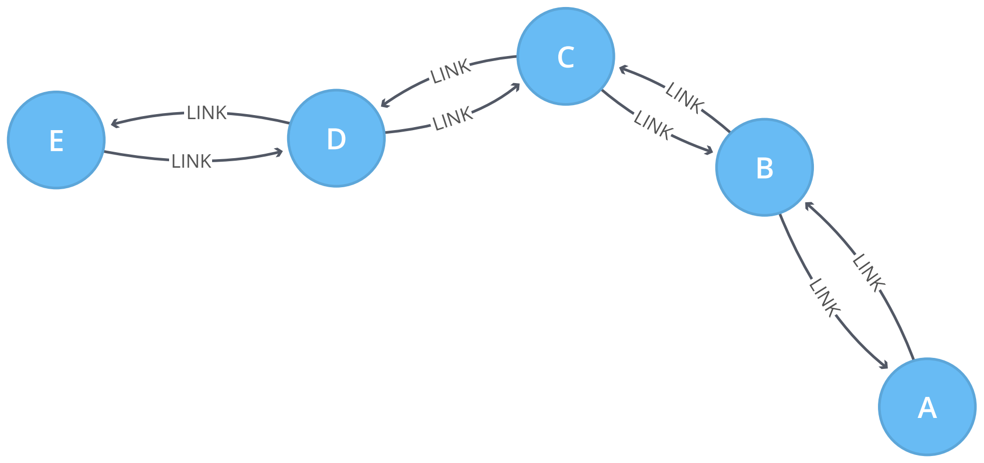
Academically, closeness centrality works best on connected graphs. If we use the original formula on an unconnected graph, we end up with an infinite distance between two nodes in separate connected components. This means that we'll end up with an infinite closeness centrality score when we sum up all the distances from that node. In practice, a variation on the original formula is used so that we don't run into these issues.

Closeness Centrality Example

Let’s see how Closeness Centrality works on a small dataset. The following Cypher statement creates a graph with nodes and links between them.

```
MERGE (a:Node{id:"A"})
MERGE (b:Node{id:"B"})
MERGE (c:Node{id:"C"})
MERGE (d:Node{id:"D"})
MERGE (e:Node{id:"E"})

MERGE (a)-[:LINK]->(b)
MERGE (b)-[:LINK]->(a)
MERGE (b)-[:LINK]->(c)
MERGE (c)-[:LINK]->(b)
MERGE (c)-[:LINK]->(d)
MERGE (d)-[:LINK]->(c)
MERGE (d)-[:LINK]->(e)
MERGE (e)-[:LINK]->(d);
```



Graph Model

The following query executes the Closeness Centrality algorithm:

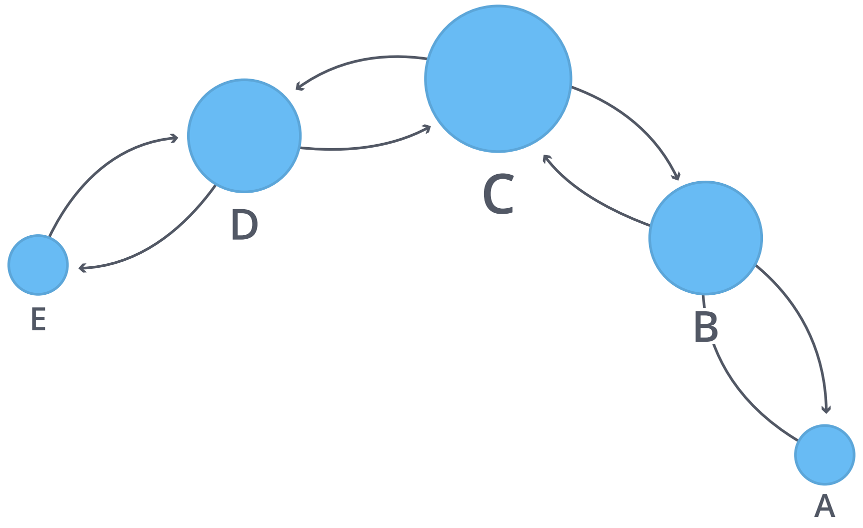
```
CALL algo.closeness.stream("Node", "LINK")
YIELD nodeId, centrality

MATCH (n:Node) WHERE id(n) = nodeId

RETURN n.id AS node, centrality
ORDER BY centrality DESC
LIMIT 20;
```

Results

Name	Centrality Weight
C	0.6666666666666666
B	0.5714285714285714
D	0.5714285714285714
A	0.4
E	0.4



Visualization of Closeness Centrality

`C` is the best connected node in this graph, although `B` and `D` aren't far behind. `A` and `E` don't have close ties to many other nodes, so their scores are lower. Any node that has a direct connection to all other nodes would score 1.

Harmonic Centrality

Harmonic Centrality (also known as valued centrality) is a variant of Closeness Centrality that was invented to solve the problem the original formula had when dealing with unconnected graphs. As with many of the centrality algorithms, it originates from the field of social network analysis.

Harmonic centrality was proposed by Marchiori and Latora in "Harmony in the Small World" while trying to come up with a sensible notion of "average shortest path."

They suggested a different way of calculating the average distance to that used in the Closeness Centrality algorithm. Rather than summing the distances of a node to all other nodes, the Harmonic Centrality algorithm sums the inverse of those distances. This enables it to deal with infinite values.

NOTE:

The raw harmonic centrality for a node is calculated using the following formula:

```
raw harmonic centrality(node) = sum(1 / distance from node to
every other node excluding itself)
```

As with Closeness Centrality we also calculate a normalized harmonic centrality with the following formula:

```
normalized harmonic centrality(node) = sum(1 / distance from node
to every other node excluding itself) / (number of nodes - 1)
```

In this formula, ∞ values are handled cleanly.

Chapter 8
Community Detection Algorithms

A fairly common feature of complex graphs is that they consist of sets of nodes that interact more with one another than with nodes outside the set. Social networks, for instance, might consist of tightly knit communities of friends with rarer friendship ties between different communities. The idea that community structures might be a defining characteristic of complex systems was first proposed by H.A. Simon in 1962.

When using community detection algorithms, we need to be conscious of the density of the relationships in the subgraphs on which we're running the algorithm. If it's very dense and all nodes are connected to each other, we may end up with all nodes congregating in one cluster. On the other hand if it's too sparse and few nodes are connected, then we may end up with each node in its own cluster.

Algorithm Type	What It Does	Example Uses
Strongly Connected Components	Locates groups of nodes where each node is reachable from every other node in the same group following the direction of relationships. It's often applied from a depth-first search.	Strongly Connected Components is often used to enable running other algorithms independently on an identified cluster. As a preprocessing step for directed graphs, it helps quickly identify disconnected groups. In retail recommendations, it helps identify groups with strong affinities that are then used for suggesting commonly preferred items to those within a given group who have not yet purchased one of the items.
Weakly Connected Components (Union Find)	Finds groups of nodes where each node is reachable from any other node in the same group, regardless of the direction of relationships. It provides near constant-time (independent of input size) operations to add new groups, merge existing groups and determine whether two nodes are in the same group.	Weakly Connected Components is often used in conjunction with other algorithms, especially for high-performance grouping. As a preprocessing step for undirected graphs, it helps quickly identify disconnected groups.
Label Propagation	Spreads labels based on neighborhood majorities as a means of inferring clusters. This extremely fast graph partitioning requires little prior information and is widely used in large-scale networks for community detection. It's a key method for understanding the organization of a graph and is often a primary step in other analysis.	Label Propagation has diverse applications from understanding consensus formation in social communities to identifying sets of proteins that are involved together in a process (functional modules) for biochemical networks. It's also used in semi- and unsupervised machine learning as an initial preprocessing step.
Louvain Modularity	Measures the quality (i.e., presumed accuracy) of a community grouping by comparing its relationship density to a suitably defined random network. It's often used to evaluate the organization of complex networks, in particular, community hierarchies. It's also useful for initial data preprocessing in unsupervised machine learning.	Louvain is used to evaluate social structures in Twitter, LinkedIn and YouTube. It's also used in fraud analytics to evaluate whether a group has just a few bad behaviors or is acting as a fraud ring that would be indicated by a higher relationship density than average. Louvain revealed a six-level customer hierarchy in a Belgian telecom network.
Triangle Count and Average Clustering Coefficient	Measures how many nodes have triangles and the degree to which nodes tend to cluster together. The average clustering coefficient is 1 when there is a clique, and 0 when there are no connections. For the clustering coefficient to be meaningful, it should be significantly higher than a version of the network where all of the relationships have been shuffled randomly.	The Average Clustering Coefficient is often used to estimate whether a network might exhibit "small-world" behaviors that are based on tightly knit clusters. It's also a factor for cluster stability and resiliency. Epidemiologists have used the Average Clustering Coefficient to help predict various infection rates for different communities.

Strongly Connected Components

The Strongly Connected Components (SCC) algorithm finds sets of connected nodes in a directed graph where each node is reachable in both directions from any other node in the same set. It is often used early in a graph analysis process to give us an idea of how our graph is structured.

SCC is one of the earliest graph algorithms, and the first linear-time algorithm was described by Tarjan in 1972. Decomposing a directed graph into its strongly connected components is a classic application of the depth-first search algorithm.

When Should I Use Strongly Connected Components?

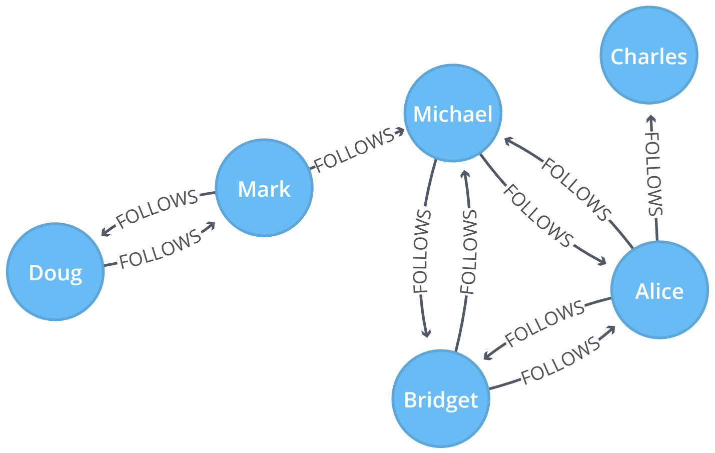
- In the analysis of powerful transnational corporations, SCC is used to find the set of firms in which every member directly owns and/or indirectly owns shares in every other member. Although it has benefits, such as reducing transaction costs and increasing trust, this type of structure weakens market competition. Read more in "The Network of Global Corporate Control."
- SCC has been used to compute the connectivity of different network configurations when measuring routing performance in multihop wireless networks. Read more in "Routing performance in the presence of unidirectional links in multihop wireless networks."
- Strongly Connected Components algorithms are often used as a first step in many graph algorithms that work only on strongly connected graphs. In social networks, a group of people are generally strongly connected (for example, students of a class or any other common place). Many people in these groups generally like some common pages or play common games. The SCC algorithms are used to find such groups and suggest the commonly liked pages or games to the people in the group who have not yet liked those pages or games.

Strongly Connected Components Example

Let's see the Strongly Connected Components algorithm in action. The following Cypher statement creates a Twitter-esque graph containing users and `FOLLOWS` relationships between them.

```
MERGE (nAlice:User {id:"Alice"})
MERGE (nBridget:User {id:"Bridget"})
MERGE (nCharles:User {id:"Charles"})
MERGE (nDoug:User {id:"Doug"})
MERGE (nMark:User {id:"Mark"})
MERGE (nMichael:User {id:"Michael"})

MERGE (nAlice)-[:FOLLOWS]->(nBridget)
MERGE (nAlice)-[:FOLLOWS]->(nCharles)
MERGE (nMark)-[:FOLLOWS]->(nDoug)
MERGE (nMark)-[:FOLLOWS]->(nMichael)
MERGE (nBridget)-[:FOLLOWS]->(nMichael)
MERGE (nDoug)-[:FOLLOWS]->(nMark)
MERGE (nMichael)-[:FOLLOWS]->(nAlice)
MERGE (nAlice)-[:FOLLOWS]->(nMichael)
MERGE (nBridget)-[:FOLLOWS]->(nAlice)
MERGE (nMichael)-[:FOLLOWS]->(nBridget);
```



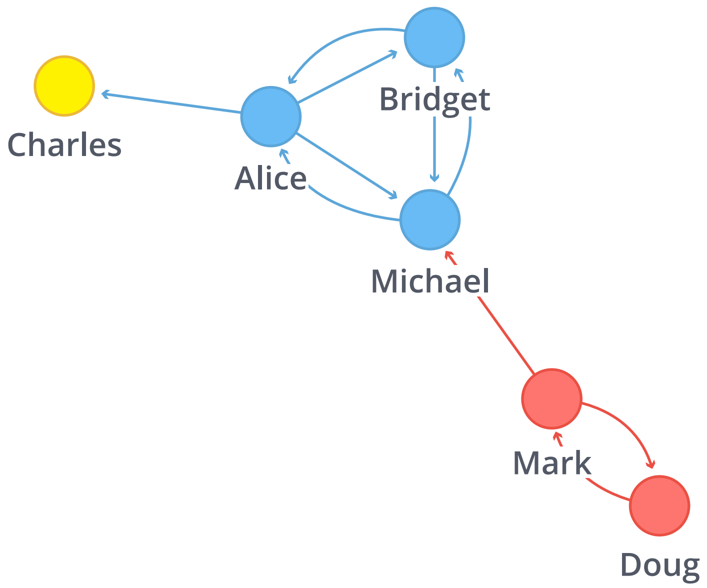
Graph Model

Now we can run Strongly Connected Components to see whether everybody is connected to each other. Execute the following query.

```
CALL algo.scc.stream("User","FOLLOWS")
YIELD nodeId, partition
MATCH (u:User) WHERE id(u) = nodeId
RETURN u.id AS name, partition
```

Results

Name	Partition
Alice	1
Bridget	1
Michael	1
Charles	0
Doug	2
Mark	2



Visualization of Strongly Connected Components

We have three strongly connected components in our sample graph.

The first, and biggest, component has members Alice, Bridget, and Michael, while the second component has Doug and Mark. Charles ends up in his own component because there isn't an outgoing relationship from that node to any of the others.

Weakly Connected Components (Union Find)

The Weakly Connected Components, or Union Find, algorithm finds sets of connected nodes in an undirected graph where each node is reachable from any other node in the same set. It differs from the Strongly Connected Components algorithm (SCC) because it only needs a path to exist between pairs of nodes in one direction, whereas SCC needs a path to exist in both directions. As with SCC, Union Find is often used early in an analysis to understand a graph's structure.

Bernard A. Galler and Michael J. Fischer first described this algorithm in 1964. The components in a graph are computed using either the breadth-first search or depth-first search algorithms.

When Should I Use Union Find?

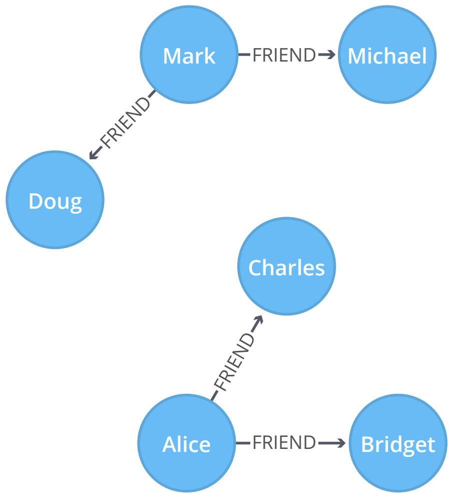
- Testing whether a graph is connected is an essential pre-processing step for every graph algorithm. Such tests are performed so quickly and easily that you should always verify that your input graph is connected, even when you know it has to be. Subtle, difficult-to-detect bugs often result when your algorithm is run only on one component of a disconnected graph.
- Union Find is also used to keep track of clusters of database records, as part of the de-duplication process – an important task in master data management applications. Read more in "An Efficient Domain-Independent Algorithm for Detecting Approximately Duplicate Database Records."
- Weakly Connected Components (WCC) is used to analyze citation networks as well. One study uses WCC to work out how well-connected the network is, and then to see whether the connectivity remains if "hub" or "authority" nodes are moved from the graph. Read more in "Characterizing and Mining Citation Graph of Computer Science Literature."

Union Find Example

Let's see the Union Find algorithm in action. The following Cypher statement creates a graph of people and their friends.

```
MERGE (nAlice:User {id:"Alice"})
MERGE (nBridget:User {id:"Bridget"})
MERGE (nCharles:User {id:"Charles"})
MERGE (nDoug:User {id:"Doug"})
MERGE (nMark:User {id:"Mark"})
MERGE (nMichael:User {id:"Michael"})

MERGE (nAlice)-[:FRIEND]->(nBridget)
MERGE (nAlice)-[:FRIEND]->(nCharles)
MERGE (nMark)-[:FRIEND]->(nDoug)
MERGE (nMark)-[:FRIEND]->(nMichael);
```



Graph Model

Now we run Union Find to find connected components. Execute the following query.

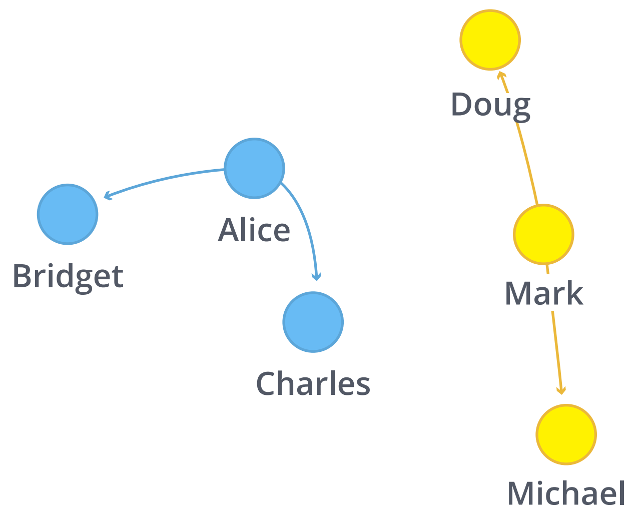
```
CALL algo.unionFind.stream("User", "FRIEND", {})
YIELD nodeId,setId

MATCH (u:User) WHERE id(u) = nodeId

RETURN u.id AS user, setId
```

Results

Name	Centrality Weight
C	0.6666666666666666
B	0.5714285714285714
D	0.5714285714285714
A	0.4
E	0.4



Visualization of Union Find

We have two distinct groups of users that have no link between them.

The first group contains Alice, Charles and Bridget, while the second group contains Michael, Doug and Mark.

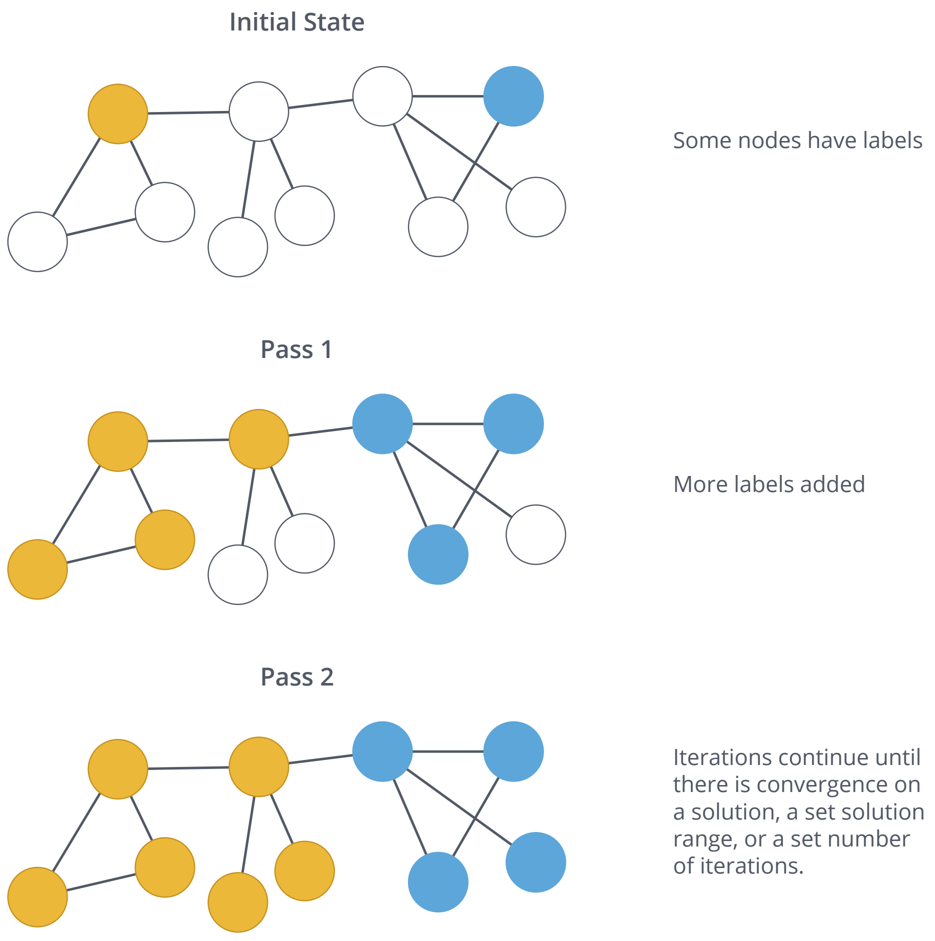
Label Propagation

The Label Propagation algorithm (LPA) is a fast algorithm for finding communities in a graph. It detects these communities using network structure alone as its guide and doesn't require a predefined objective function or prior information about the communities.

One interesting feature of LPA is that you have the option of assigning preliminary labels to narrow down the range of generated solutions. This means you can use it as a semi-supervised way of finding communities where you handpick some initial communities.

LPA is a relatively new algorithm and was only proposed by Raghavan et al. in 2007, in "Near linear time algorithm to detect community structures in large-scale networks." It works by propagating labels throughout the network and forming communities based on this process of label propagation.

The intuition behind the algorithm is that a single label can quickly become dominant in a densely connected group of nodes, but it will have trouble crossing a sparsely connected region. Labels will get trapped inside a densely connected group of nodes, and those nodes that end up with the same label when the algorithm finishes are considered part of the same community.


Initial State
Some nodes have labels
Pass 1
More labels added
Pass 2
Iterations continue until there is convergence on a solution, a set solution range, or a set number of iterations.

Label Propagation Algorithm

NOTE: The algorithm works as follows:

- Every node is initialized with a unique label (an identifier).
- These labels propagate through the network.
- At every iteration of propagation, each node updates its label to the one that the maximum number of its neighbors belongs to. Ties are broken uniformly and randomly.
- LPA reaches convergence when each node has the majority label of its neighbors.

As labels propagate, densely connected groups of nodes quickly reach a consensus on a unique label. At the end of the propagation, only a few labels will remain – most will have disappeared. Nodes that have the same label at convergence are said to belong to the same community.

When Should I Use Label Propagation?

- Label Propagation has been used to assign polarity of tweets, as a part of semantic analysis that uses seed labels from a classifier trained to detect positive and negative emoticons in combination with the Twitter follower graph. For more information, see "Twitter polarity classification with label propagation over lexical links and the follower graph."
- Label Propagation has been used to estimate potentially dangerous combinations of drugs to co-prescribe to a patient, based on the chemical similarity and side effect profiles. The study is found in "Label Propagation Prediction of Drug-Drug Interactions Based on Clinical Side Effects."
- Label Propagation has been used to infer features of utterances in a dialogue for a machine learning model to track user intention with the help of a Wikidata knowledge graph of concepts and their relations. For more information, see "Feature Inference Based on Label Propagation on Wikidata Graph for DST."

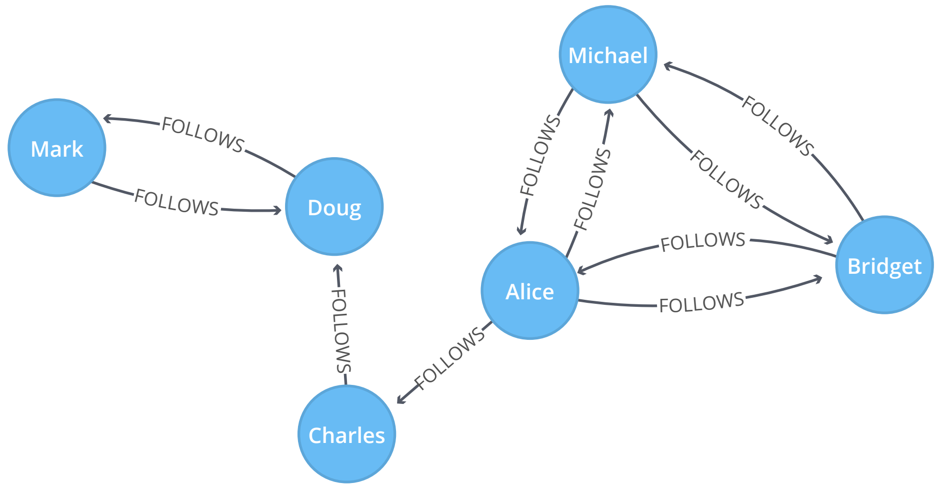
TIP: In contrast with other algorithms, Label Propagation results in different community structures when run multiple times on the same graph. The range of solutions is narrowed if some nodes are given preliminary labels, while others are unlabeled. Unlabeled nodes are more likely to adopt the preliminary labels.

Label Propagation Example

Let's see the Label Propagation algorithm in action. The following Cypher statement creates a Twitter-esque graph containing users and `FOLLOWS` relationships between them.

```
MERGE (nAlice:User {id:"Alice"})
MERGE (nBridget:User {id:"Bridget"})
MERGE (nCharles:User {id:"Charles"})
MERGE (nDoug:User {id:"Doug"})
MERGE (nMark:User {id:"Mark"})
MERGE (nMichael:User {id:"Michael"})

MERGE (nAlice)-[:FOLLOWS]->(nBridget)
MERGE (nAlice)-[:FOLLOWS]->(nCharles)
MERGE (nMark)-[:FOLLOWS]->(nDoug)
MERGE (nBridget)-[:FOLLOWS]->(nMichael)
MERGE (nDoug)-[:FOLLOWS]->(nMark)
MERGE (nMichael)-[:FOLLOWS]->(nAlice)
MERGE (nAlice)-[:FOLLOWS]->(nMichael)
MERGE (nBridget)-[:FOLLOWS]->(nAlice)
MERGE (nMichael)-[:FOLLOWS]->(nBridget)
MERGE (nCharles)-[:FOLLOWS]->(nDoug);
```



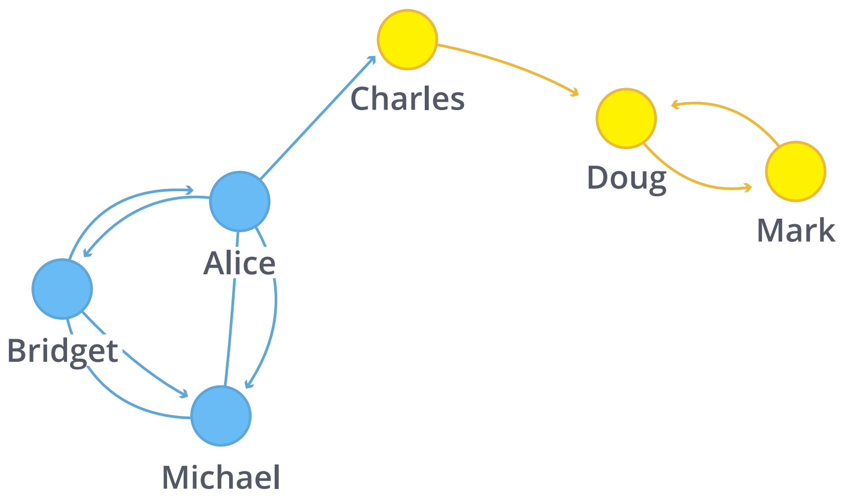
Graph Model

Now we run LPA to find communities among the users. Execute the following query.

```
CALL algo.labelPropagation.stream("User", "FOLLOWS",
  {direction: "OUTGOING", iterations: 10})
```

Results

Name	Partition
Alice	5
Charles	4
Bridget	5
Michael	5
Doug	4
Mark	4



Visualization of Label Propagation

Our algorithm found two communities with three members each.

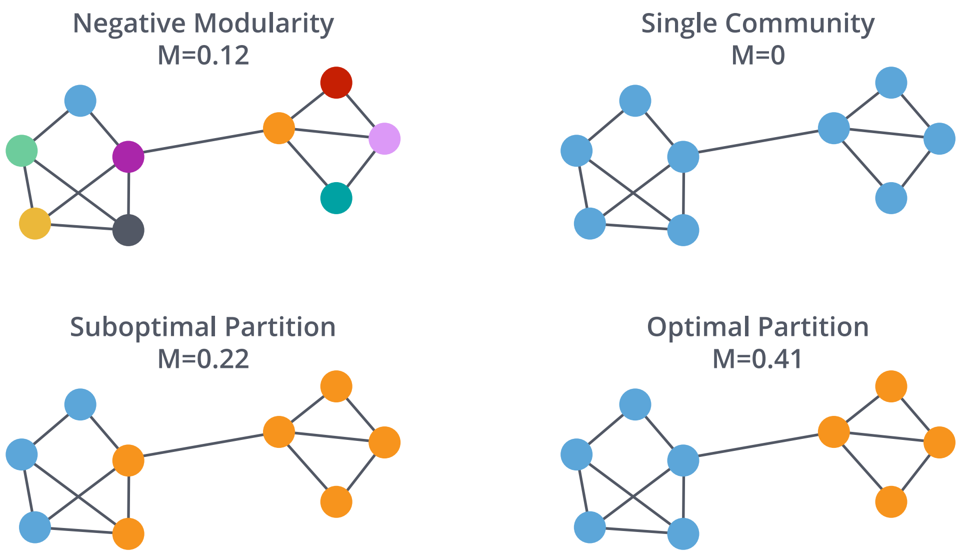
It appears that Michael, Bridget and Alice belong together, as do Doug and Mark. Only Charles doesn't strongly fit into either side, but ends up with Doug and Mark.

Louvain Modularity

The Louvain method of community detection is an algorithm for detecting communities in networks. It maximizes a modularity score for each community, where the modularity quantifies the quality of an assignment of nodes to communities by evaluating how much more densely connected the nodes within a community are, compared to how connected they would be in a random network.

The Louvain algorithm is one of the fastest modularity-based algorithms and works well with large graphs. It also reveals a hierarchy of communities at different scales, which is useful for understanding the global functioning of a network.

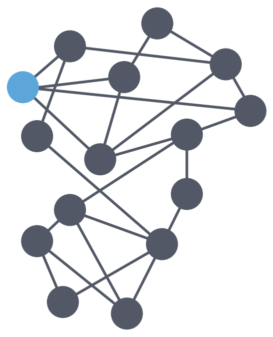
In order to understand the Louvain modularity algorithm, we must first look at modularity in general.



Modularity

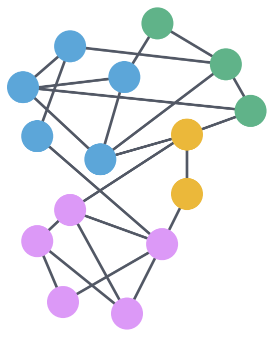
Modularity is a measure of how well groups have been partitioned into clusters. It compares the relationships in a cluster compared to what would be expected for a random (or other baseline) number of connections.

Pass 1



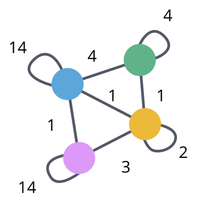
Step 0

Choose a start node and calculate the change in modularity that would occur if that node joins and forms a community with each of its immediate neighbors.



Step 1

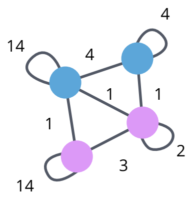
The start node joins the node with the highest modularity change. The process is repeated for each node with the above communities formed.



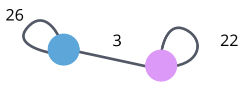
Step 2

Communities are aggregated to create super communities and the relationships between these super nodes are weighted as a sum of previous links. (Self-loops represent the previous relationships now hidden in the super node.)

Pass 2



Step 1



Step 2

Steps 1 and 2 repeat in passes until there is no further increase in modularity or a set number of iterations have occurred.

Louvain Modularity Algorithm

The Louvain algorithm was proposed in 2008. The method consists of repeated application of two steps. The first step is a "greedy" assignment of nodes to communities, favoring local optimizations of modularity. The second step is the definition of a new coarse-grained network based on the communities found in the first step. These two steps are repeated until no further modularity-increasing reassignments of communities are possible.

When Should I Use Louvain?

- The Louvain method has been proposed to provide recommendations for Reddit users to find similar subreddits based on general user behavior. For more details, see "Subreddit Recommendations within Reddit Communities."
- The Louvain method has been used to extract topics from online social platforms, such as Twitter and YouTube, based on the co-occurence graph of terms in documents as a part of the topic modeling process. This process is described in "Topic Modeling based on Louvain method in Online Social Networks."
- The Louvain method has been used to investigate the human brain and find hierarchical community structures within the brain's functional network. The study mentioned is "Hierarchical Modularity in Human Brain Functional Networks."

TIP: Although the Louvain method, and modularity optimization algorithms more generally, have found wide applications across many domains, some problems with these algorithms have been identified:

1. The *resolution* limit

For larger networks, the Louvain method doesn't stop with the "intuitive" communities. Instead, there's a second pass through the community modification and coarse-graining stages, in which several of the intuitive communities are merged together. This is a general problem with modularity optimization algorithms; they have trouble detecting small communities in large networks. It's a virtue of the Louvain method that something close to the intuitive community structure is available as an intermediate step in the process.

2. The *degeneracy* problem

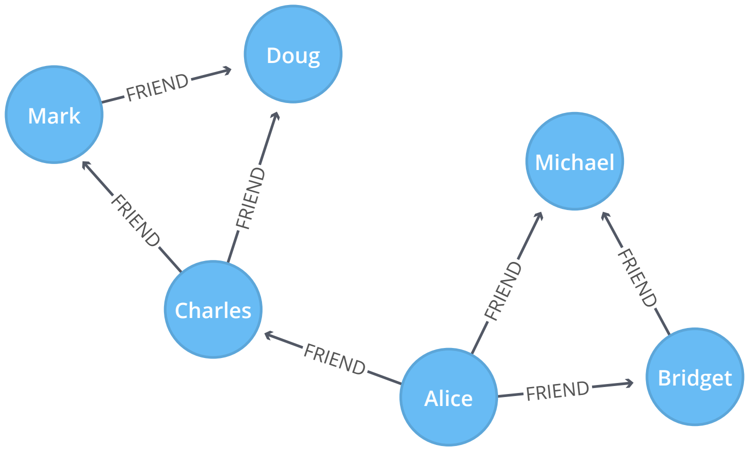
There is typically an exponentially large (in network size) number of community assignments with modularities close to the maximum. This can be a severe problem because, in the presence of a large number of high modularity solutions, it's hard to find the global maximum and difficult to determine if the global maximum is truly more scientifically important than local maxima that achieve similar modularity. Research shows that the different locally optimal community assignments have different structural properties. For more information, see "The performance of modularity maximization in practical contexts."

Louvain Example

Let's see the Louvain algorithm in action. The following Cypher statement creates a graph of users and friends.

```
MERGE (nAlice:User {id:"Alice"})
MERGE (nBridget:User {id:"Bridget"})
MERGE (nCharles:User {id:"Charles"})
MERGE (nDoug:User {id:"Doug"})
MERGE (nMark:User {id:"Mark"})
MERGE (nMichael:User {id:"Michael"})

MERGE (nAlice)-[:FRIEND]->(nBridget)
MERGE (nAlice)-[:FRIEND]->(nCharles)
MERGE (nMark)-[:FRIEND]->(nDoug)
MERGE (nBridget)-[:FRIEND]->(nMichael)
MERGE (nCharles)-[:FRIEND]->(nMark)
MERGE (nAlice)-[:FRIEND]->(nMichael)
MERGE (nCharles)-[:FRIEND]->(nDoug);
```



Graph Model

Now we run Louvain to find communities in the social network. Execute the following query.

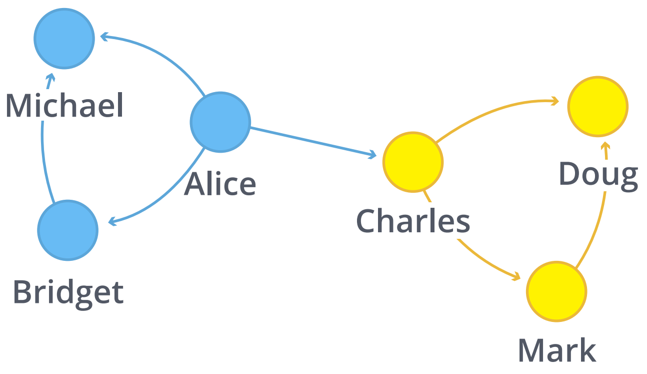
```
CALL algo.louvain.stream("User", "FRIEND", {})
YIELD nodeId, community

MATCH (user:User) WHERE id(user) = nodeId

RETURN user.id AS user, community
ORDER BY community;
```

Results

Name	Partition
Alice	5
Bridget	5
Michael	5
Charles	4
Doug	4
Mark	4



Visualization of Louvain

Our algorithm found two communities with three members each.

Mark, Doug and Charles are all friends with each other, as are Bridget, Alice and Michael. Charles is the only one who has friends in both communities, but he has more in community four so he fits better in that one.

Triangle Count and Clustering Coefficient

Triangle Count is a community detection graph algorithm that is used to determine the number of triangles passing through each node in the graph. A triangle is a set of three nodes, where each node has a relationship to all other nodes.

Triangle counting gained popularity in social network analysis, where it is used to detect communities and measure the cohesiveness of those communities. It is also used to determine the stability of a graph and is often used as part of the computation of network indices, such as the clustering coefficient.

There are two types of clustering coefficients:

Local clustering coefficient
The local clustering coefficient of a node is the likelihood that its neighbors are also connected. The computation of this score involves triangle counting.

Global clustering coefficient
The global clustering coefficient is the normalized sum of those local clustering coefficients.

The transitivity coefficient of a graph is sometimes used, which is three times the number of triangles divided by the number of triples in the graph. For more information, see "Finding, Counting and Listing all Triangles in Large Graphs, An Experimental Study."

When Should I Use Triangle Count and Clustering Coefficient?

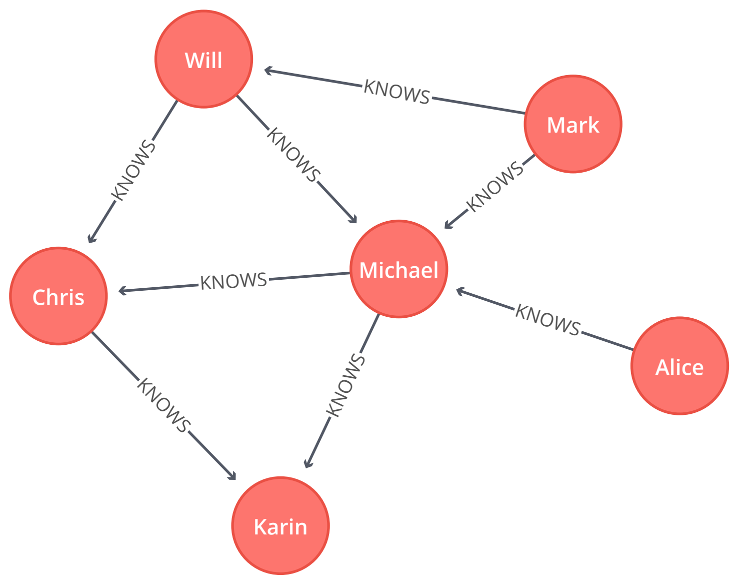
- Triangle Count and Clustering Coefficient have been shown to be useful as features for classifying a given website as spam or non-spam content. This is described in "Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs."
- Clustering Coefficient has been used to investigate the community structure of Facebook's social graph, where they found dense neighborhoods of users in an otherwise sparse global graph. Find this study in "The Anatomy of the Facebook Social Graph."
- Clustering Coefficient has been proposed to help explore the thematic structure of the Web and detect communities of pages with a common topic based on the reciprocal links between them. For more information, see "Curvature of co-links uncovers hidden thematic layers in the World Wide Web."

Triangles Example

Let's see how the Triangle Count and Clustering Coefficient algorithm works on a small dataset. The following Cypher statement creates a graph with people and `KNOWS` relationships between them.

```
MERGE (alice:Person{id:"Alice"})
MERGE (michael:Person{id:"Michael"})
MERGE (karin:Person{id:"Karin"})
MERGE (chris:Person{id:"Chris"})
MERGE (will:Person{id:"Will"})
MERGE (mark:Person{id:"Mark"})

MERGE (michael)-[:KNOWS]->(karin)
MERGE (michael)-[:KNOWS]->(chris)
MERGE (will)-[:KNOWS]->(michael)
MERGE (mark)-[:KNOWS]->(michael)
MERGE (mark)-[:KNOWS]->(will)
MERGE (alice)-[:KNOWS]->(michael)
MERGE (will)-[:KNOWS]->(chris)
MERGE (chris)-[:KNOWS]->(karin);
```



Graph Model

The following query finds all the `KNOWS` triangles between people in our graph.

```
CALL algo.triangle.stream("Person","KNOWS")
YIELD nodeA,nodeB,nodeC

MATCH (a:Person) WHERE id(a) = nodeA
MATCH (b:Person) WHERE id(b) = nodeB
MATCH (c:Person) WHERE id(c) = nodeC

RETURN a.id AS nodeA, b.id AS nodeB, c.id AS node
```

Results

NodeA	NodeB	NodeC
Will	Michael	Chris
Will	Mark	Michael
Michael	Karin	Chris

We can see that there are `KNOWS` triangles containing "Will, Michael and Chris", "Will, Mark and Michael", and "Michael, Karin and Chris." This means that everybody in the triangle knows each other.

We work out the clustering coefficient of each person by running the following algorithm.

```
CALL algo.triangleCount.stream('Person', 'KNOWS')
YIELD nodeId, triangles, coefficient

MATCH (p:Person) WHERE id(p) = nodeId

RETURN p.id AS name, triangles, coefficient
ORDER BY coefficient DESC
```

Results

Name	Triangles	Coefficient
Karin	1	1
Mark	1	1
Chris	2	0.6666666666666666
Will	2	0.6666666666666666
Michael	3	0.3
Alice	0	0

We learn that Michael is part of the most triangles, but it's Karin and Mark who are the best at introducing their friends – all of the people who know them, know each other!

We've covered a lot of ground so far and learned about lots of different graph algorithms. In the next chapter, we'll take things a step further and see how to glue everything together using a real-world dataset.

Chapter 9
Graph Algorithms in Practice

In this section we'll learn how to apply graph algorithms in data-intensive applications. We will be using data from Yelp's annual dataset challenge.

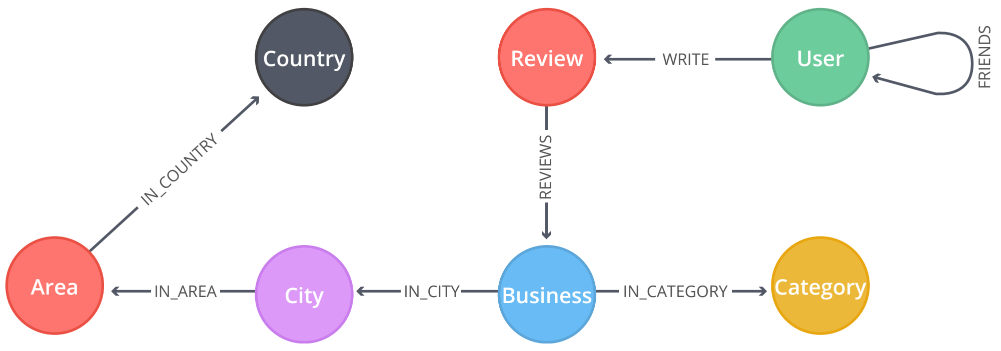
Yelp.com has been running the Yelp Dataset challenge since 2013, a competition that encourages people to explore and research Yelp's open dataset. As of Round 10 of the challenge, the dataset contained:

- Almost 5 million reviews
- Over 1.1 million users
- Over 150,000 businesses
- 12 metropolitan areas

Since its launch, the dataset has become popular, with hundreds of academic papers written about it. It has well-structured and highly interconnected data and is therefore a realistic dataset with which to showcase Neo4j and graph algorithms.

Graph Model

The Yelp data is represented in a graph model as shown in the diagram below.



Yelp Graph Model

Our graph contains `User` labeled nodes, which have a `FRIENDS` relationship with other `Users`. `Users` also `WRITE Reviews` and tips about `Businesses`. All of the metadata is stored as properties of nodes, except for `Categories` of the `Businesses`, which are represented by separate nodes.

Data Import

There are many different methods for importing data into Neo4j, including the import tool, LOAD CSV command and Neo4j Drivers.

For the Yelp dataset we need to do a one-off import of a large amount of data so the import tool is the best choice. See the yelp-graph-algorithms GitHub repository for more details.

Exploratory Data Analysis

Once we have the data loaded in Neo4j, we execute some exploratory queries to get a feel for it. We will be using the Awesome Procedures on Cypher (APOC) library in this section. Please see the "Installing APOC" section in Appendix B for more details.

The following queries return the cardinalities of node labels and relationship types.

```
CALL db.labels()
YIELD label
CALL apoc.cypher.run("MATCH (:`"+label+"`) RETURN count(*) as count", null)
YIELD value
RETURN label, value.count as count
ORDER BY label
```

Results

Label	Count
Area	54
Business	174567
Category	1293
City	1093
Country	17
Review	5261669
User	1326101

```
CALL db.relationshipTypes()
YIELD relationshipType
CALL apoc.cypher.run("MATCH ()-[:" + `relationshipType` + "]->()
                      RETURN count(*) as count", null)
YIELD value
RETURN relationshipType, value.count AS count
ORDER BY relationshipType
```

Results

Label	Count
FRIENDS	10645356
IN_AREA	1154
IN_CATEGORY	667527
IN_CITY	174566
IN_COUNTRY	54
REVIEWS	5261669
WROTE	5261669

These queries shouldn't reveal anything surprising but they are useful for checking that the data has been imported correctly.

It's always fun reading hotel reviews, so we're going to focus on businesses in that sector. We find out how many hotels there are by running the following query.

```
MATCH (category:Category {name: "Hotels"})
RETURN size((category)<-[:IN_CATEGORY]-()) AS businesses
```

Results

Hotels
2683

That's a decent number of hotels to explore.

How many reviews do we have to work with?

```
MATCH (:Review)-[:REVIEWS]->(:Business)-[:IN_CATEGORY]->(:Category {name:"Hotels"})
RETURN count(*) AS count
```

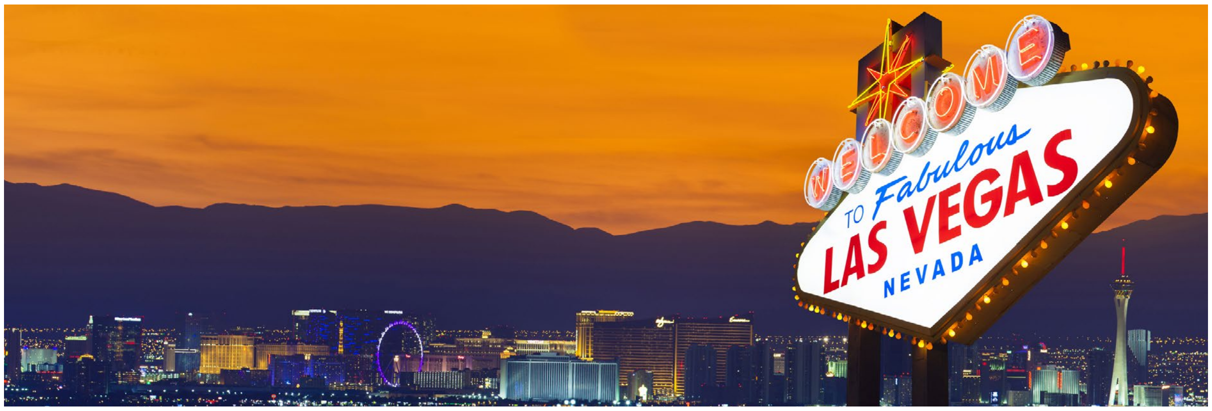
Results

Count
183759

Let's zoom in on some of the individual bits of data.

Trip Planning

Imagine that we're planning a trip to Las Vegas and want to find somewhere to stay.



We might start by asking which are the most reviewed hotels and how well they've been rated.

```
MATCH (review:Review)-[:REVIEWS]->(business:Business),
      (business)-[:IN_CATEGORY]->(:Category {name:"Hotels"}),
      (business)-[:IN_CITY]->(:City {name: "Las Vegas"})
WITH business, count(*) AS reviews, avg(review.stars) AS averageRating
ORDER BY reviews DESC
LIMIT 10
RETURN business.name AS business,
       reviews,
       apoc.math.round(averageRating,2) AS averageRating
```

Results

Hotel	Reviews	Average Rating
ARIA Resort & Casino	3794	3.51
The Cosmopolitan of Las Vegas	3772	3.87
Luxor Hotel and Casino Las Vegas	3623	2.63
MGM Grand Hotel	3445	2.99
The Venetian Las Vegas	3103	3.93
Flamingo Las Vegas Hotel & Casino	2942	2.48
Bellagio Hotel	2781	3.71
Mandalay Bay Resort & Casino	2688	3.27
Planet Hollywood Las Vegas Resort & Casino	2682	3.05
Monte Carlo Hotel And Casino	2506	2.64

These hotels have **a lot** of reviews, far more than anyone would be likely to read. We'd like to find the best reviews and make them more prominent on our business page.

Finding Influential Hotel Reviewers

One way we can do this is by ordering reviews based on the influence of the reviewer on Yelp.

We'll start by finding users who have reviewed more than five hotels. After that we'll find the social network between those users and work out which users sit at the center of that network. This should reveal the most influential people. The `FRIENDS` relationship is an example of a bidirectional relationship, meaning that if Person A is friends with Person B then Person B is also friends with Person A. Neo4j stores a directed graph, but we have the option to ignore the direction when we query the graph.

We want to execute the PageRank algorithm over a projected graph of users that have reviewed hotels and then add a `hotelPageRank` property to each of those users. This is the first example where we can't express the projected graph in terms of node labels and relationship types. Instead we will write Cypher statements to project the required graph.

See the "Usage" section of Chapter 5 for a refresher on the different usage options.

The following query executes the PageRank algorithm.

```
CALL algo.pageRank(
    "MATCH (u:User)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->
           (:Category {name: "Hotels"})
     WITH u, count(*) AS reviews
     WHERE reviews > 5
     RETURN id(u) AS id",
    "MATCH (u1:User)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->
           (:Category {name: "Hotels"})
     MATCH (u1)-[:FRIENDS]->(u2)
     WHERE id(u1) < id(u2)
     RETURN id(u1) AS source, id(u2) AS target",
    {graph: "cypher", write: true, direction: "both", writeProperty: "hotelPageRank"}
)
```

We then write the following query to find the top reviewers.

```
MATCH (u:User)
WHERE u.hotelPageRank > 0
WITH u
ORDER BY u.hotelPageRank DESC
LIMIT 5
RETURN u.name AS name,
       apoc.math.round(u.hotelPageRank,2) AS pageRank,
       size((u)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->
            (:Category {name: "Hotels"})) AS hotelReviews,
       size((u)-[:WROTE]->()) AS totalReviews,
       size((u)-[:FRIENDS]-()) AS friends
```

Results

name	pageRank	hotelReviews	totalReview	Friends
Jason	17.93	7	60	5159
Jamie	14.59	8	41	688
Jeremy	11.57	6	28	6164
Lori	9.9	6	39	4518
Connie	7.98	7	51	5336

We could use those rankings on a hotel page when determining which reviews to show first. For example, if we want to show reviews of Caesars Palace, we could execute the following query.

```
MATCH (b:Business {name: "Caesars Palace Las Vegas Hotel & Casino"})
MATCH (b)<-[:REVIEWS]-(review)<-[:WROTE]-(user)
RETURN user.name AS name,
       apoc.math.round(user.hotelPageRank,2) AS pageRank,
       review.stars AS stars
ORDER BY user.hotelPageRank DESC
LIMIT 5
```

Results

name	pageRank	stars
Jason	17.93	3
Amanda	7.28	4
J	6.88	4
Michelle	4.73	4
Pasquale	4.58	3

This information may also be useful for businesses that want to know when an influencer is staying in their hotel.

Finding Similar Categories

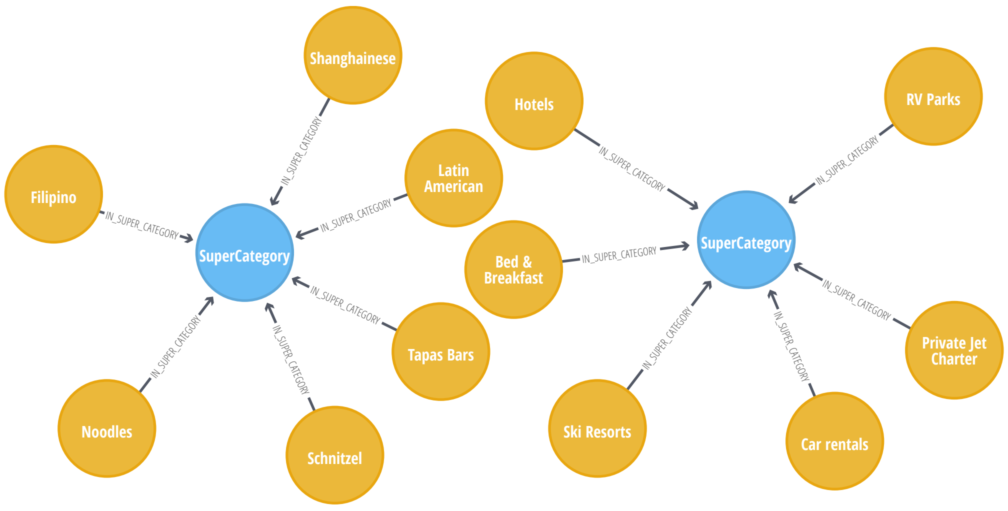
The Yelp dataset contains more than 1,000 categories, and it seems likely that some of those categories are similar to each other. That similarity is useful for making recommendations to users for other businesses that they may be interested in.

We will build a weighted category similarity graph based on how businesses categorize themselves. For example, if only one business categorizes itself under `Hotels and Historical Tours`, then we would have a link between `Hotels and Historical Tours` with a weight of 1.

We don't actually have to create the similarity graph – we can run a community detection algorithm, such as Label Propagation, over a projected similarity graph.

```
CALL algo.labelPropagation.stream(
  "MATCH (c:Category) RETURN id(c) AS id",
  "MATCH (c1:Category)<-[:IN_CATEGORY]-()-[:IN_CATEGORY]->(c2:Category)
   WHERE id(c1) < id(c2)
   RETURN id(c1) AS source, id(c2) AS target, count(*) AS weight",
   {graph: "cypher"})
YIELD nodeId, label
MATCH (c:Category) WHERE id(c) = nodeId
MERGE (sc:SuperCategory {name: "SuperCategory-" + label})
MERGE (c)-[:IN_SUPER_CATEGORY]->(sc
```

The diagram below shows a sample of categories and super categories after we've run this query.



Super Categories

We write the following query to find some of the similar categories to hotels.

```
MATCH (hotels:Category {name: "Hotels"}),
      (hotels)-[:IN_SUPER_CATEGORY]->()<-[:IN_SUPER_CATEGORY]-(otherCategory)
RETURN otherCategory.name AS otherCategory
LIMIT 5
```

Results

otherCategory
Bed & Breakfast
Private Jet Charter
Ski Resorts
Car Rental
RV Parks
Motorcycle Rental
Bus Rental
Scooter Tours
Historical Tours
Trains

Not all of those categories are relevant for users in Las Vegas, so we need to write a more specific query to find the most popular similar categories in this location.

```
MATCH (hotels:Category {name: "Hotels"}),
      (lasVegas:City {name: "Las Vegas"}),
      (hotels)-[:IN_SUPER_CATEGORY]->()<-[:IN_SUPER_CATEGORY]-(otherCategory)
RETURN otherCategory.name AS otherCategory,
       size((otherCategory)<-[:IN_CATEGORY]-()-[:IN_CITY]->(lasVegas)) AS count
ORDER BY count DESC
LIMIT 10
```

Results

otherCategory	count
Event Planning & Services	1608
Venues & Event Spaces	228
Insurance	211
Tours	189
Transportation	176
Car Rental	160
Travel Services	96
Limos	84
Resorts	73
Airport Shuttles	52

We could then make a suggestion of one business with an above average rating in each of those categories.

```
MATCH (hotels:Category {name: "Hotels"}),
      (lasVegas:City {name: "Las Vegas"}),
      (hotels)-[:IN_SUPER_CATEGORY]->()<-[:IN_SUPER_CATEGORY]-(otherCategory),
      (otherCategory)<-[:IN_CATEGORY]-(business)-[:IN_CITY]->(lasVegas)
WITH otherCategory, count(*) AS count,
     collect(business) AS businesses,
     apoc.coll.avg(collect(business.averageStars)) AS categoryAverageStars
ORDER BY count DESC
LIMIT 10
WITH otherCategory,
     [b in businesses where b.averageStars >= categoryAverageStars] AS businesses
RETURN otherCategory.name AS otherCategory,
       [b in businesses | b.name][toInteger(rand() * size(businesses))] AS business
```

Results

otherCategory	business
Event Planning & Services	Viva Las Vegastamps
Venues & Event Spaces	VIP Golf Services
Insurance	Desert Shores Insurance Services
Tours	Annie Bananie Las Vegas Tours
Transportation	Sinderella Coach
Car Rental	Hertz Rent A Car
Travel Services	MW Travel Vegas
Limos	Vegas Limousine Service
Resorts	Encore
Airport Shuttles	Presidential Limousine

In this chapter, we've shown just a couple of ways that insights from graph algorithms are used in a real-time workflow to make real-time recommendations. In our example we made category and business recommendations but graph algorithms are applicable to many other problems.

Graph algorithms can help you take your graph-powered application to the next level.

Conclusion

Graph analytics have value only if you have the skills to use them and if they can quickly provide the insights you need. Graph algorithms are easy to use, fast to execute and produce powerful results.

Graph algorithms are the powerhouse behind the analysis of real-world networks – from identifying fraud rings and optimizing the location of public services to evaluating the strength of a group and predicting the spread of disease or ideas.

In this ebook, you've learned about how graph algorithms help you make sense of connected data. We covered the types of graph algorithms and offered specifics about how to use each one. Still, we are keenly aware that we have only scratched the surface. There is so much more to learn. Check out the Neo4j Graph Data Science Library. If you have any questions or need any help with any of the material in this ebook, send us an email at devrel@neo4j.com. We look forward to hearing how you are using graph algorithms.

Learn More

- Neo4j Graph Algorithms Documentation
- Awesome Procedures on Cypher
- Graph Algorithms Sandbox
- Graph Algorithms Jupyter Notebooks
- Graph Algorithms Webinar
- Graph Algorithms Overview White Paper

Appendix A: Performance Testing

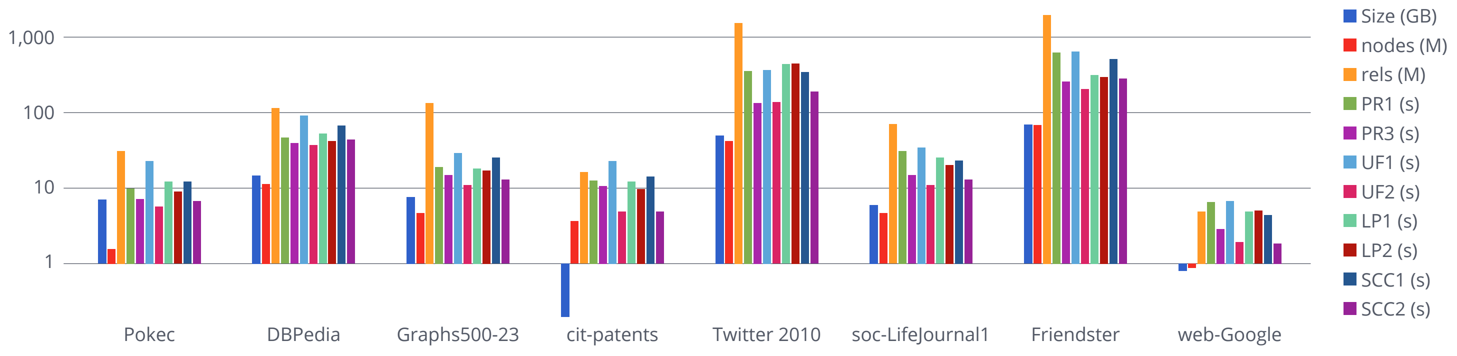
For PageRank, Union Find, Label Propagation and Strongly Connected Components, we have run preliminary tests on medium and larger datasets that have been used in other publications.

The table contains database size and node and relationship counts. For each algorithm you see the runtime (in seconds) of the first and second run.

Comparing them with other publications these runtimes look good, but of course the real confirmation comes from you running the algorithms on your own datasets and hardware.

Graph	Size (GB)	nodes (M)	rels (M)	PR1 (s)	PR20 (s)	UF1 (s)	UF2 (s)	LP1 (s)	LP2 (s)	SCC1 (s)	SCC2 (s)
Pokec	7.3	2	31	10	7	24	6	12	9	12	7
DBPedia	15	11	117	46	38	91	37	51	43	65	41
Graphs500-23	7.9	5	129	19	15	29	10	18	17	25	13
cit-patents	0.2	4	17	13	10	23	5	12	10	14	5
Twitter-2010	49	42	1468	349	131	353	128	405	405	339	189
soc-LifeJournal1	6.3	5	69	30	14	34	11	25	19	23	13
Friendster	62	66	1806	611	235	619	196	296	282	483	257

Below is a log-scale showing the same data in one chart.



Appendix B: Installing the Neo4j Graph Algorithms Library

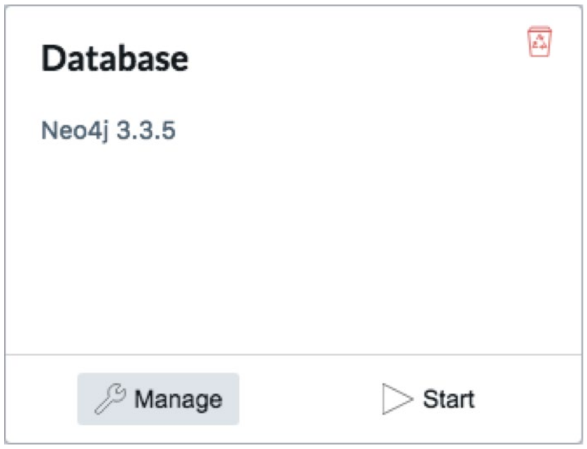
Appendix B contains instructions for installing the tools and libraries referenced in this ebook.

Installing Neo4j Desktop

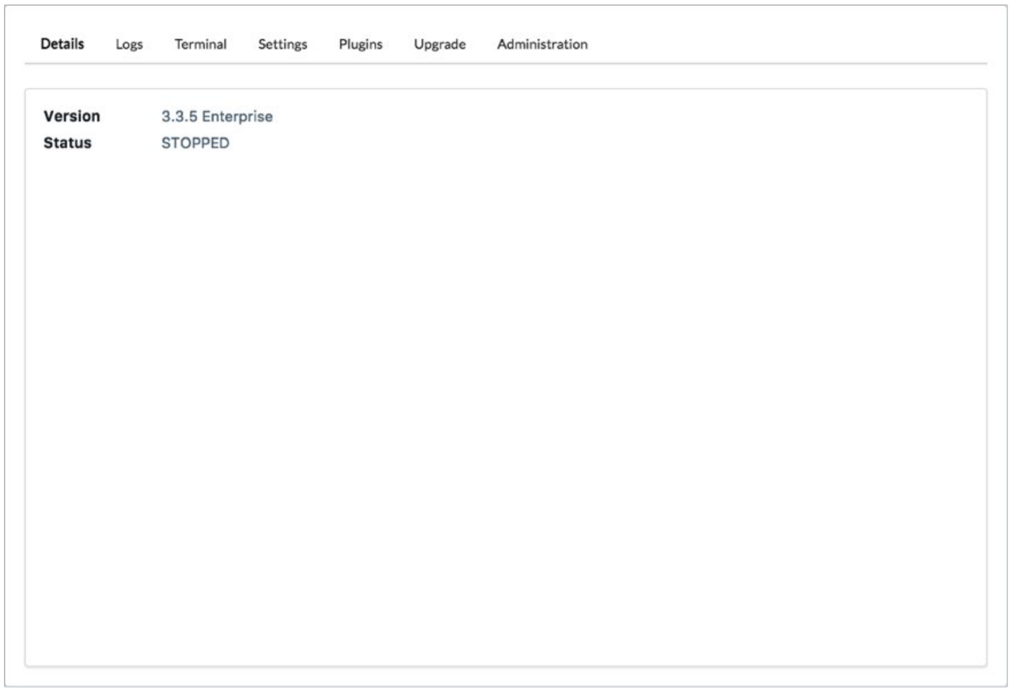
Download Neo4j Desktop from neo4j.com/download. After you've installed it, follow the instructions to create a project and corresponding database.

Installing Graph Algorithms

Once you've installed Neo4j Desktop and created your first project, click on the "Manage" button for your database.

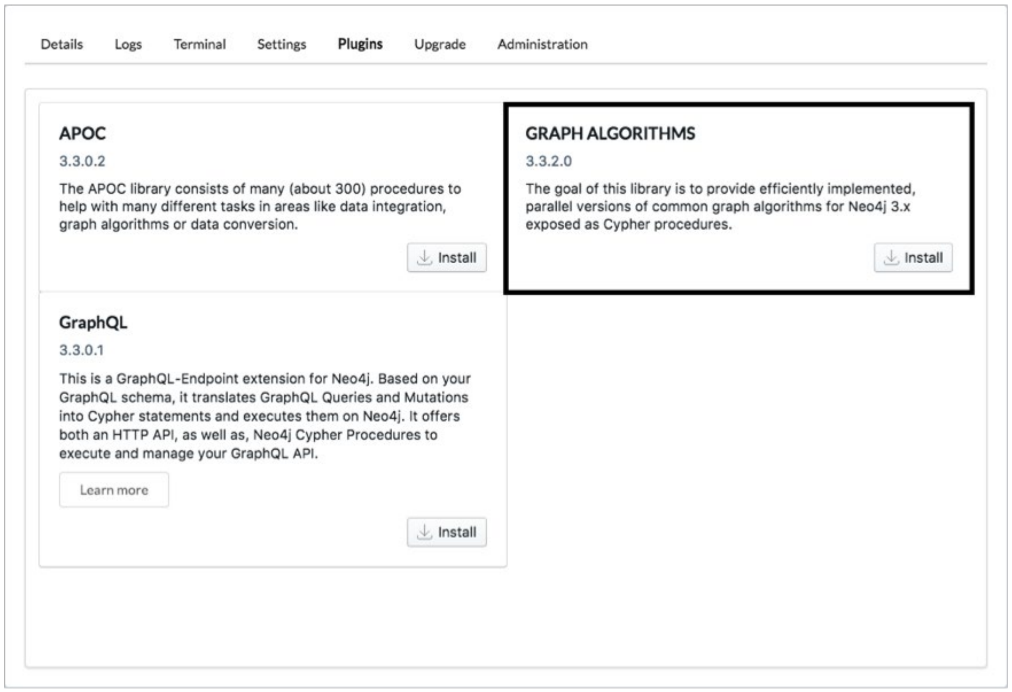


You will see this screen (note that the plugins can only be installed when the database is not running).



Version 3.3.5 Enterprise
Status STOPPED

Click on the "Plugins" button and you will see this screen.



APOC

3.3.0.2

The APOC library consists of many (about 300) procedures to help with many different tasks in areas like data integration, graph algorithms or data conversion.

GRAPH ALGORITHMS

3.3.2.0

The goal of this library is to provide efficiently implemented, parallel versions of common graph algorithms for Neo4j 3.x exposed as Cypher procedures.

GraphQL

3.3.0.1

This is a GraphQL-Endpoint extension for Neo4j. Based on your GraphQL schema, it translates GraphQL Queries and Mutations into Cypher statements and executes them on Neo4j. It offers both an HTTP API, as well as, Neo4j Cypher Procedures to execute and manage your GraphQL API.

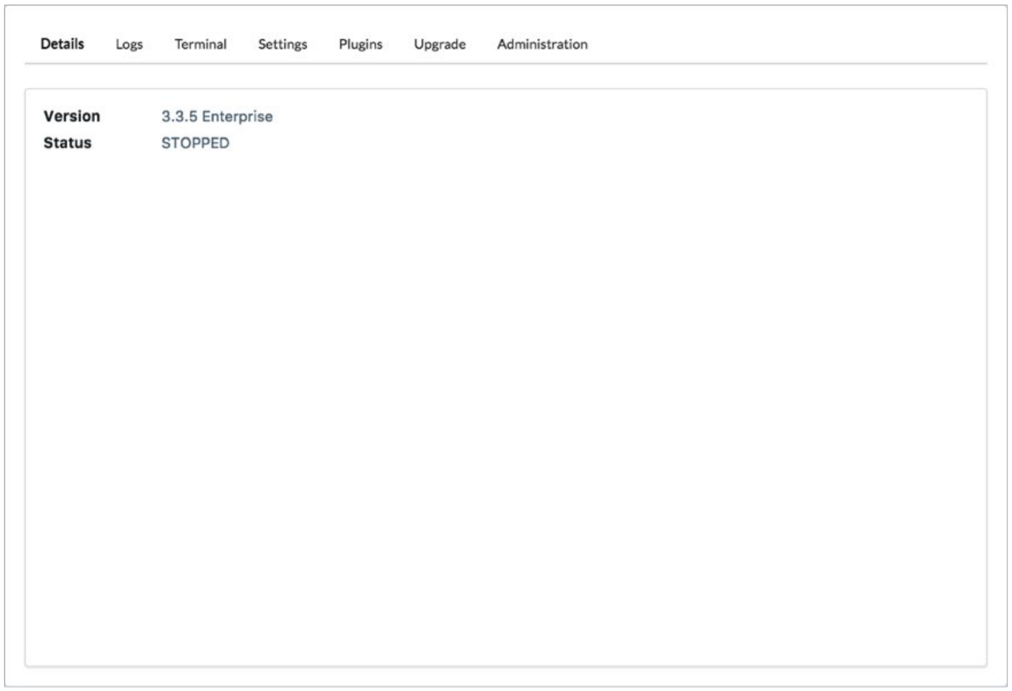
Click on the "Install" button for the graph algorithms plugin and wait for the plugin to be installed. It may take a few seconds depending on your internet bandwidth. The database will be restarted to allow the plugin to be picked up.

Installing APOC

The APOC (Awesome Procedures on Cypher) library consists of procedures and functions to help with many different tasks in areas such as data integration and data conversion.

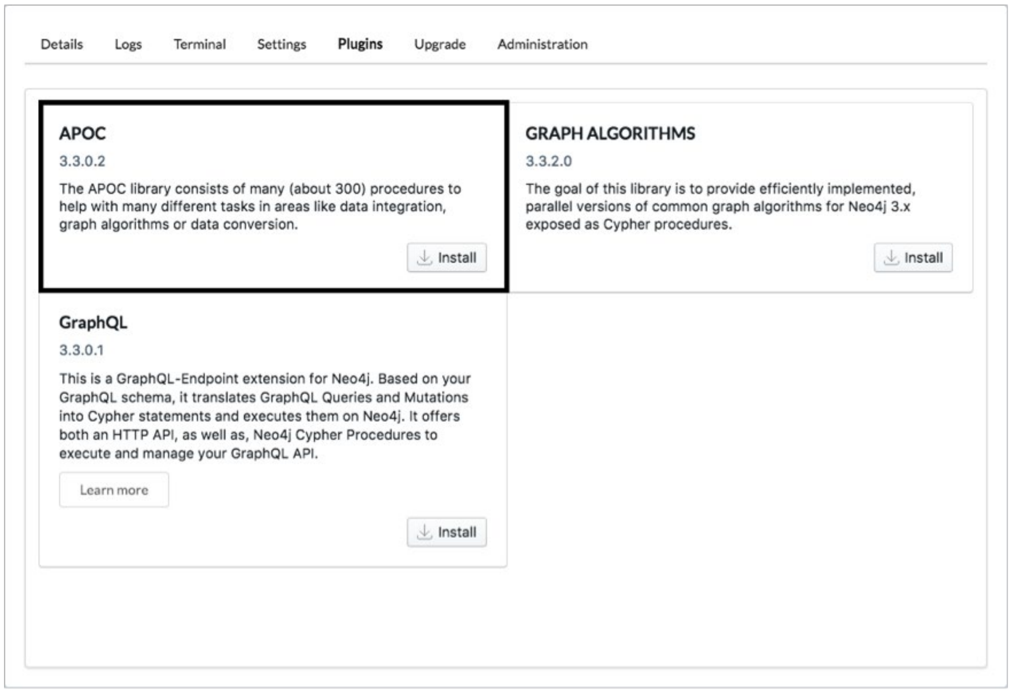
Once you've installed Neo4j Desktop and created your first project, click on the "Manage" button for your database.

You will see this screen.



Version	3.3.5 Enterprise
Status	STOPPED

Click on the "Plugins" button and you will see this screen.



APOC
3.3.0.2
The APOC library consists of many (about 300) procedures to help with many different tasks in areas like data integration, graph algorithms or data conversion.
Install

GRAPH ALGORITHMS
3.3.2.0
The goal of this library is to provide efficiently implemented, parallel versions of common graph algorithms for Neo4j 3.x exposed as Cypher procedures.
Install

GraphQL
3.3.0.1
This is a GraphQL-Endpoint extension for Neo4j. Based on your GraphQL schema, it translates GraphQL Queries and Mutations into Cypher statements and executes them on Neo4j. It offers both an HTTP API, as well as, Neo4j Cypher Procedures to execute and manage your GraphQL API.
Learn more
Install

Click on the "Install" button for the APOC plugin and wait for the plugin to be installed. It may take a few seconds depending on your internet bandwidth. The database will be restarted to allow the plugin to be picked up.