Understanding Trolls with Efficient Analytics of Large Graphs in Neo4j

David Allen[1], Amy E. Hodler[2], Michael Hunger[3], Martin Knobloch[4], William Lyon[5], Mark Needham[6], Hannes Voigt[7]

Abstract: Analytics of large graph data set has become an important means of understanding and influencing the world. The use of graph database technology in the International Consortium of Investigative Journalists' (ICIJ) investigation of the Panama Papers and Paradise Papers or in cancer research illustrates how analysing graph-structured data helps to uncover important but hidden relationships. A very current example in that regards shows how graph analytics can help shed light on the operations of social media troll-networks, e.g. on Twitter. In similar fashion, graph analytics can help enterprises to unearth hidden patterns and structures within connected data, to make more accurate predictions and faster decisions. All this requires efficient graph analytics well-integrated with management of graph data.

The Neo4j Graph Platform provides such an environment. It provides transactional processing and analytical processing of graph data including data management and analytics tooling. A central element for graph analytics in the Graph Platform are the Neo4j graph algorithms. Neo4j graph algorithms provide efficiently implemented, parallel versions of common graph algorithms, integrated and optimized for the Neo4j transactional database. In this paper, we will describe the design and integration Neo4j Graph Algorithms, demonstrate its utility of our approach with a Twitter Troll analysis, and show case its performance with a few experiments on large graphs.

Keywords: Graph databases, graph analytics, graph algorithms, property graphs

1 Introduction

While we perceive ourselves as individuals, it is undeniable, that we are also highly-connected to everything around us. This contrast of individuals and connections give rise to two approaches of perceiving and understanding the world. The *entity-centered* approach focuses on the individual, it describes the world by primarily characterizing and specifying individual entities. A classical example of entity-centered is Aristotle's descriptions of

[1] Neo4j, San Mateo, US david.allen@neo4j.com
[2] Neo4j, San Mateo, US amy.hodler@neo4j.com
[3] Neo4j, Dresden, Germany michael.hunger@neo4j.com
[4] Avantgarde Labs, Dresden, Germany mknobloch@avantgarde-labs.de
[5] Neo4j, San Mateo, US william.lyon@neo4j.com
[6] Neo4j, London, UK mark.needham@neo4j.com
[7] Neo4j, Leipzig, Germany hannes.voigt@neo4j.com

doi:10.18420/btw2019-23

species by their parts such as live birth, number of legs, etc. during his stay on the Island of Lesbos. The *relationship-centered* approach, in contrast, focuses on relationships, it describes the world by primarily characterizing and specifying the relationships between individual entities. For instance, the world of the living can be further understood by looking at relationships among living such as prey–predator and evolutionary relationships.

We are connected to many other individuals, events and things that populate the world around us, near and far. Connectedness manifests in many different ways, facets, values, scopes, and scales. To understand reality or parts of it, we have to understand the connectedness that pervades it, the connectedness that forms its fabric. Because of the multifacet, multiscale nature of connectedness, the precise connections that help our understanding are not necessarily apparent, though. Often the interesting connections are intricately hidden in between many other, different layers of connectedness, or only apparent in transitive paths spanning many intermediate steps. A classical example of this is a conflict-of-interest connection of a person with a process the person has a decision power over. A conflict-of-interest can easily behind an intricate path of affiliation, proximity, relationship, ownership, coercion, bribery, etc. connections. Other examples are understanding drug-target efficacy in drug design and environmental impact of genetically modified organisms.

Today's ubiquity of information technology makes increasing amounts of connected data from many different sources accessible within an organization and outside of it. Such data becomes available in various data models. Conceptually the simplest data model of expressing and describing connected information is the property graph model [RN10]. A property graph is relationship-centered, it has relationships a first-class citizens. Entities carry, per instance, arbitrary properties and optional labels to describe their roles. Next to all the entities of consideration, a property graph explicitly supports the connections these entities have and – what sets it apart from other common graph data models – describes connections as rich relationships, with a type and arbitrary properties. A single property graph can capture whole variety of the connectedness existing in the world and makes it easy for humans to reason about very big complicated phenomena and their combined interactions.

Pulling multiple different data source of connections together in a property graph and analysing patterns of connection by means of graph theory and network analysis often reveals hidden relationships and structures, characterizes and quantifies them, and provides new understanding. Two main ingredients to make such an analysis happening are capabilities for managing graph data and run graph analysis algorithms efficiently even on very large graphs. Neo4j's graph platform provides these capabilities, among others. In particular, (1) the graph platform offers ETL and data integration functionality to extract data connections from various data formats and moving them into a single graph database. (2) The graph platform provides high-performance graph algorithms for analysis, that are well integrated with management capabilities of graph database.

In this paper we provide an overview of the graph platform (Sect. 2) and give a detailed presentation of the capabilities, design, and architecture of the graph algorithms (Sect. 3). We demonstrate its utility of our approach with a Twitter Troll analysis (Sect. 4), and show case its performance with a few experiments on large graphs (Sect. 5). Finally, we discuss related approaches (Sect. 6) and conclude the paper (Sect. 7).

2 Graph Platform

The graph platform is a well integrated set of tools, surfaces and infrastructure for storing, managing and querying connected data. At the frontend, the graph platform offers a variety of skill-specific, user-facing tools and APIs for solution developers, data scientists, and business users. Each user group has different needs to extract data from various sources, execute statements, explore query results, update information, perform analytical steps, and visualize connections. The comprehensive set of well-integrated tools on top of a common protocol, API, and query language provides efficient means for data scientists and solutions teams to move through the stages of discovery and design.

To make graph data equally accessible to tabular information, a query language that is able to express the richness of graph patterns in an comprehensive way is a powerful tool for developer and end-users alike. With Neo4j's declarative query language Cypher [Fr18], which uses graph pattern represented in ascii-art-symbols for pattern matching, creation, predicates and more, even complex conceptual connections can be expressed. With additional full support for list and map data structures and means processing those, many round trips of traditional query languages can be avoided. Built in data-flow paradigms allow users to clearly express more involved processing pipelines.

The graph platform is extensible with user defined procedures and functions, which are able to access the full scope of the underlying API and machine infrastructure, while being exposed as Cypher clauses or expressions. This extension mechanism was also used in our subsequently described work.

At the backend, the graph platform provides for both analytic and transactional operations. All operations leverage the efficient traversal of connections provided by the graph store and embedded in a scalable architecture. The graph platform can scale up to 32 TB of memory space enabling in-memory and near-memory graph processing on an ultra-large scale in a single machine. The graph platform can also scale out to multiple machines with Raft protocol-based [OO14] causal clustering architecture [HA90, Ah95] that supports ultra-large clusters and a wider range of cluster topologies for distributed data centers and cloud.

3 Graph Algorithms

Neo4j graph algorithms is a library of user-defined procedures that can be executed on a Neo4j database. The analysis procedures provide efficient parallel implementation of common graph analysis algorithms. The library covers algorithms for path finding (e.g. all pair shortest path and minimum weight spanning tree), centrality analysis (e.g. parge rank and betweenness centrality), and community detection (e.g. label propagation and louvain modularity), as of now. The procedures can be called directly using Cypher in the Neo4j Browser, from the cypher shell, from Jupyter notebooks or any other client code. For running these algorithms a (projected) (sub-)graph of data is concurrently extracted from Neo4j into a separate, in-memory graph-storage to provide isolation and high read-only operations performance. Then the appropriate algorithm is executed concurrently on that graph storage, taking graph structure and additional information such as node- or relationship-weights into account. After the computation finished, the results can either be streamed to the client or optionally written back efficiently in a concurrent manner, e.g. to node-properties or relationships.

The general call syntax to call a analysis procedure is:

```
CALL algo.<name>(<nodeSelector>, <relSelector>, {<config>})
YIELD <columnList>
```

Here, `<name>` specifies the procedure to call, `<nodeSelector>` and `<relSelector>` specify the graph of interest, `<config>` provide additional parameters of the procedure, and `<columnList>` specifies, which column of output of the procedures should be returned. Most algorithms provide graph processing statistics and time measurements as well as statistical summaries of the computed graph metrics (e.g. min, max, avg, stdev, and percentiles of a centrality). An alternative variant of each algorithm is available as `algo.<name>.stream` which instead streams back the algorithm results in multiple columns of data. That volume would be equivalent to the size of the graph of interest, potentially billions of rows.

Process: All analysis procedure follow the same three-step process. (1) Load the graph of interest in parallel from the database into a succinct in-memory data structures. (2) Run graph algorithm in parallel. (3) Consume results. If multiple procedures should be executed, the graph of interest can be pre-loaded once and then referred to by name.

3.1 Graph Loading

Typically, the property graph managed in the database contains much more information than relevant for a certain analysis step. In such case, the (sub)graph of interest is a projection of the managed property graph. The projected graph is either a directed graph or undirected graph with the possibility of node and relationship weights. The projected graph is not multigraph (like a property graph) and allows only a single edge between a pair of nodes in each direction.

Projection: The graph algorithms library provides two kinds of projection: (1) label-based projection and (2) Cypher-based projection. Label-based projection extracts all nodes with a label given as `<nodeSelector>` and all relationships of a type given as `<relSelector>`. For instance,

```
CALL algo.pageRank('Page', 'LINKS')
```

calls the procedure `pageRank` on nodes with label `Page` and relationships of type `LINKS`. If either or both are left off then the algorithm will run on all entities of the respective, connected type.

In contrast, Cypher-based projection extracts a subgraph based on two Cypher queries. The first query given as `<nodeSelector>` specifies the nodes of interest as a node-id-list and the second query given as `<relSelector>` specifies the relationship of interest as a list of start-end-node-ids. For instance,

```
CALL algo.pageRank(
  "MATCH (u:User) WHERE exists( (u)-[:FRIENDS]-() ) RETURN id(u) AS id",
  "MATCH (u1:User)-[:FRIENDS]-(u2:User)
   RETURN id(u1) AS source, id(u2) AS target",
  {graph:"cypher"}
)
```

calls the procedure `pageRank` on a graph containing (1) nodes that match the pattern `(u:User)` and have at least one `FRIENDS` relationship and (2) relationships of type `FRIENDS` between two nodes with label `User`. Note that relationships that do not have both source and target nodes described by the `<nodeSelector>` will be ignored. This projection is presenting a *graph view* over the existing data and can be used to contract or aggregate the graph or collapse intermediate paths. For instance, it can be used to render a user-to-user graph from a social network via intermediate mentions, jointly used tags or replies on messages. Or it can provide category or product similarity graphs based on joint reviews or purchases by users. With the Cypher-based projection, any n-partite graph can be projected to a mono-partite graph for the means of running graph analysis.

Efficient Loading: To quickly load the relevant sub-graphs from Neo4j into the dedicated data structures, the library uses low level APIs of the graph database to avoid memory churn and preferably only use primitive numeric types. During the loading the id-space of the original graph is remapped to a consecutive id-space in the algorithm space to allow for gap free operations and iteration. The load operations are executed in parallel with the given concurrency, each thread operating on a batch of graph records at a time. For graph projections via Cypher we utilize the compiled runtime that turns Cypher queries into efficient Java bytecode. There we use Cypher's pagination capabilities to distribute the loading across concurrent threads.

Graph representation: The graph algorithm library uses a dedicated in-memory representation of the projected graph data. This allows for isolation from the transactional graph, for holding arbitrary graph projections and for high-performance random access. These data structures are presented with a thin, numeric-id based API to each algorithm, which allows for different implementations. Different aspects of the graph API are separated and can be passed individually to an algorithm according to its needs. The projected, potentially huge graph is stored in-memory in a succinct way. Particularly, all nodes are mapped to a compressed, dense integer domain. The mapping is stored (1) in a dense array used to map the dense integer domain back to the original node ids and (2) in a sparse array to map the original node ids forward to the dense integer domain.

Relationships without weights are stored in a CSR-like structure [Bu09]. A source node id-indexed, paged array holds the position of each node's adjacency in a second array. Target node ids are delta and variable length encoded per adjacency. For relationships with weights the adjacency of a node is represented with two arrays, one for the target nodes and one for the weights. Depending on the algorithm's needs, the only incoming edge are presented or the adjacency is additionally replicated as outgoing edges. The representation is designed to scale to very large graphs of billions of edges. The loader allocates only the structures needed, e.g. weight entries are not created for default values or not at all if node or relationship weights are not needed.

After an algorithm is finished using the graph structure it is able to release that memory to make it available for further processing.

3.2 Algorithm Execution

Tab. 1 lists all algorithms currently available as procedure in Neo4j graph algorithms. There are algorithms for centrality measuring, path finding, and community detection. For each algorithm, we reference work on which its implementation is based on.

Path Finding	All-Pair Shortest Path	[Di59, KS91]
	K Shortest Path	[Ye70, Ye71]
	Single-Source Shortest Path	[MS03, Ma07]
	K Spanning Tree	[Pr57]
	Minimum Spanning Tree	[NMN01]
	A* Shortest Path	[HNR68]
Centrality Measuring	Betweenness Centrality	[Fr77, Br01, BP07, Ko13]
	Closeness Centrality	[Ba50]
	Dangalchev Centrality	[Da06]
	Harmonic Centrality	[ML00]
	PageRank	[MR05, GZB04]
Community Detection	Label Propagation	[RAK07, FI14]
	Louvain Modularity	[Bl08, Wi14]
	Triangle Counting and Clustering Coefficient	[Ts08, CC11]
	Strongly Connected Components	[Ta72, MNS17, SRM14]
	Balanced Triads	[He58, Fl63]

Tab. 1: Graph analytical algorithms implemented in Neo4j graph algorithms.

Each algorithm utilizes the provided graph representation to optimally batch and execute partial operations concurrently. The partial operations are orchestrated by a concurrency infrastructure that handles utilization and work redistribution. This allows threads that finish earlier to pick up work from threads computing metrics for nodes with a higher degree. For the implementation of the algorithms we evaluated and implemented the latest state of the art research with dedicated focus on parallelization and scaling.

Each stage of loading, algorithm execution and result production tracks and reports memory consumption to allow configuration adjustments for future invocations.

3.3 Result Consumptions

The library offers three kinds of result consumptions: (1) *write back results*, (2) *tabular aggregated results*, and (3) *tabular streamed results*.

Write back results: The non-streaming procedures, write back results to the property graph in the database as node properties or relationships. This write back uses again low level transactional APIs and batches updates of larger chunks of the graph (e.g. 100 000 updates per transaction). By utilizing Neo4j's transaction-co-commit and joint-flush mechanics, it can achieve write performance of at least 1 M record updates per second, depending on available compute resources and I/O bandwidth.

The non-streaming procedures are indicated by the procedure name not having the suffix `.stream`. The parameter write indicates if only statistics should be computed or actual write back executed. For instance,

```
CALL algo.pageRank('Page', 'LINKS',
                   { iterations: 20, dampingFactor: 0.85,
                     write: true, writeProperty: "pagerank" })
```

calls the procedure `pageRank` with four config parameters. Two, `iterations: 20` and `dampingFactor: 0.85` are PageRank algorithm-specific, while `write: true` triggers a write back result and `writeProperty: "pagerank"` specifies the name of the node property to which the result shall be written back. In this case, the PageRank scores calculated for each node is written to the property `pagerank` of that node. Those written back graph metrics are used to enrich the original graph data and can be used afterwards by regular, OLTP query processing, e.g. for recommendations, ranking, or other decision making.

Tabular aggregated results: Procedures report the various statistics for the computed metrics and for operations (projected graph size, timings, memory usage), which the user can utilize for further processing and monitoring.

Tabular streamed results: Most algorithms can return tabular results also as a tuples stream. The procedure implementing the stream variant of an algorithm is suffixed with the name `.stream`, e.g. `algo.pageRank.stream(...)`. Tabular results are return as a driving table for further processing to the query that contains the `CALL` statement and potentially to the client. In this mode for each node entry in the projected graph one or more computed metrics are returned. For instance centrality measures, shortest paths, triangle counts, and specific triangle triples. This computed information is proportional to the size of the projected graph, i.e. it can reach billions of result rows produced.

All columns that should be returned must be explicitly selected in the YIELD clause after the CALL. Which columns are provided is documented for every procedure and can also be inspected interactively. The driving table can be further processed within Cypher or returned to the user with a RETURN clause. For instance, the query

```
CALL algo.pageRank.stream('Page', 'LINKS',
                          { iterations: 20, dampingFactor: 0.85 })
YIELD nodeId, score
RETURN nodeId, score ORDER BY score DESC LIMIT 5
```

calls the procedure algo.pageRank.stream and postfilters its result to return only the ids and scores of the top five page-ranked nodes.

Summary: The Neo4j graph algorithms procedure library provides a comprehensive set of graph analysis algorithms. The algorithms are implemented for parallel execution and operate on succinct optimized in-memory data structures that scale to very large graphs of interest. The procedures take care of data extraction and result routing. They blend in very well with Cypher. Cypher queries can call the procedures, specify the graph of interest, and consume the result for further processing in a seamless, composable fashion.

4 Analyzing Troll Behavior

To demonstrate usefulness and practicality we show how Twitter troll behavior can be analyzed and better understood with the help of Neo4j graph algorithms. As part of the House Intelligence Committee investigation into how Russia may have influenced the 2016 US Election, Twitter published the screen names of almost 3000 Twitter accounts believed to be connected to Russia's Internet Research Agency, a company known for operating social media troll accounts. Twitter immediately suspended these accounts, deleting their data from Twitter.com and the Twitter API. A team at NBC News including Ben Popken and EJ Fox was able to reconstruct a dataset consisting of a subset of the deleted data for their investigation and using Neo4j, were able to show how these troll accounts went on attack during key election moments. NBC News open-sourced the reconstructed dataset and released it as a Neo4j database. In this section we show how this analysis was conducted using Neo4j graph algorithms.

4.1 Preparation

Graph data schema: The Neo4j database was constructed from several anonymous sources who had been collecting election related tweets leading up to the 2016 US Election. These

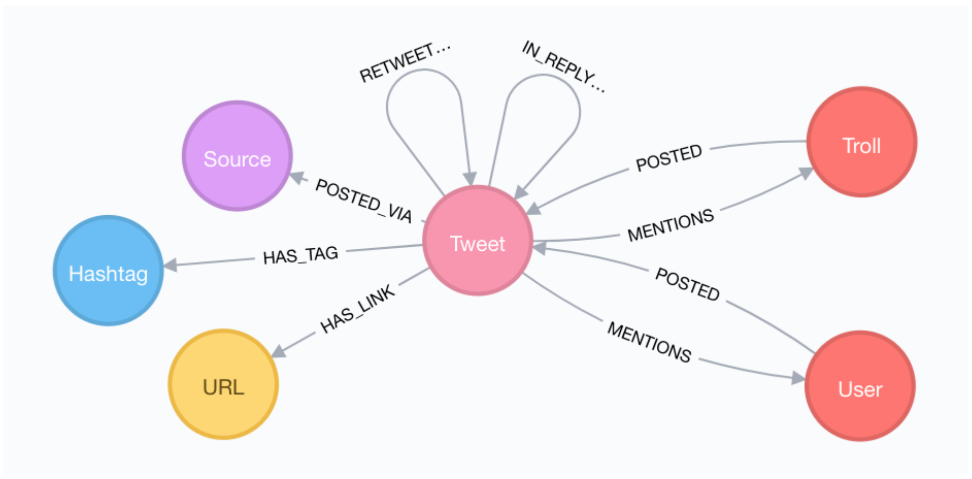


Fig. 1: The property graph schema used to represent the Russian Twitter Trolls dataset in Neo4j.

datasets were combined and imported into Neo4j so that the data is structured as shown in Fig. 1. The nodes are `Tweets`, `Users` (which can also be `Trolls`), `Hashtags`, `Sources` (the application used to post a `Tweet`), and `URLs` mentioned in `Tweets`. Relationships connect these nodes, showing for example, users mentioned in tweets and hashtags used. `User` nodes representing known troll accounts are additionally labeled `Troll`.

Exploratory data analysis with Cypher: Once the data was modeled and imported into Neo4j, Cypher queries permit exploration. Find tweets with the hashtag #thanksobama:

```
MATCH (u:User)-[:POSTED]->(t:Tweet)
        -[:HAS_TAG]->(ht:Hashtag {tag: "thanksobama"})
RETURN * LIMIT 50
```

Hashtags were used by the trolls to insert themselves into conversations and gain visibility. We can query the most common hashtags used by the trolls:

```
MATCH (ht:Hashtag)<-[:HAS_TAG]-(tw:Tweet)<-[:POSTED]-(:Troll)
WITH ht, count(tw) AS num ORDER BY num DESC
RETURN ht.tag AS hashtag, num LIMIT 10
```

One of the findings reported by the NBC team was that troll tweet volume spiked during key election related events. We can see that much of the tweet volume occurs leading up to and immediately following the 2016 US Election. Here we query for tweet volume by day:

```
MATCH (:Troll)-[:POSTED]->(t:Tweet)
WHERE t.created_str > "2016-10-01"
RETURN substring(t.created_str,0,10) AS day, count(t) AS num
  ORDER BY day LIMIT 60
```

Projected graph of interest: Much of the analysis (including the application of graph algorithms) was done on a projected monopartite `User`-to-`User` graph from the broader domain graph. For example, we consider a `Troll` account to *amplify* another `Troll` account when it posts a `Tweet` that is a retweet of a `Tweet` posted by another `Troll` account as

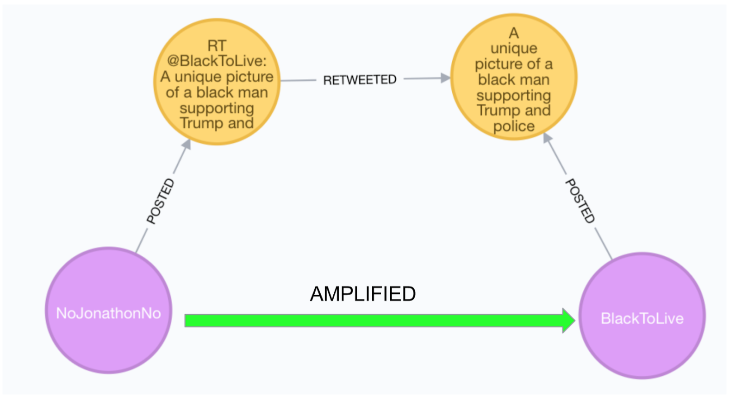

RT @BlackToLive: A unique picture of a black man supporting Trump and
RETWEETED
A unique picture of a black man supporting Trump and police
POSTED
POSTED
NoJonathonNo
AMPLIFIED
BlackToLive

Fig. 2: Illustration of the projected graph used to run graph algorithms on the Russian Twitter Trolls dataset.

illustrated in Fig. 2. We construct an projected graph of this *retweet network* for analysis. This particular analysis is based on the insight that retweets on Twitter generally indicate agreement of a message, or at least desire for broader visibility of that message, and that they also permit a network of authors to limit the amount of original content that needs to be written in order to push key themes and messages. This projected graph can be expressed using Cypher:

```
MATCH p=(r1:Troll)-[:POSTED]->(:Tweet)
           <-[:RETWEETED]-(:Tweet)<-[:POSTED]-(r2:Troll)
RETURN p LIMIT 1
```

This is however only one of the possible projections, others could be based on mentions, replies, or jointly used hashtags for topic analysis.

4.2 Analysis

For the analysis, we apply two graph algorithms. Centralities measures helps us detect who are the most influential troll accounts. Subsequently community detection helps to answer which Troll accounts are strongly connected through Retweets.

Centrality: Centrality algorithms determine the most important nodes in the network. In the context of our retweet graph, centralities will allow us to determine the most influential accounts.

We can run PageRank on the projected graph:

```
CALL algo.pageRank(
  "MATCH (t:Troll) RETURN id(t) AS id",
  "MATCH (r1:Troll)-[:POSTED]->(:Tweet)
           <-[:RETWEETED]-(:Tweet)<-[:POSTED]-(r2:Troll)
   RETURN id(r2) as source, id(r1) as target",
  {graph:"cypher", write: true, writeProperty: "pagerank"}
)
```

The above Cypher statement will write a `pagerank` property to the `Troll` nodes. We can use Cypher to find the trolls with the highest pagerank score, and are thus the most influential in the network by this measure:

```
MATCH (t:Troll) WHERE exists(t.pagerank)
RETURN t.screen_name AS troll, t.pagerank AS pagerank
  ORDER BY pagerank DESC LIMIT 25
```

With a slight modification of this query, we can assign (`SET`) each of the most influential trolls a random seeding community number:

```
MATCH (t:Troll) WHERE exists(t.pagerank)
WITH t ORDER BY t.pagerank DESC LIMIT 25
SET t.community = toInteger(round(rand()*1000))
RETURN t
```

Community detection: Community detection algorithms are applied to this projected network to partition the graph. In the context of the retweet graph community detection will illustrate which accounts are frequently amplifying others. We want to see if there are clusters of troll accounts that are promoting certain types of content. For example, are certain accounts focused on pro-Trump content while others are focused on anti-Clinton content?

Here we partition the graph into communities using the Label Propagation algorithm. Label propagation is a particularly fast algorithm for finding communities in a graph. As we are particularly interested in communities around the most influential trolls, we use the algorithm in a semi-supervised way and seed it with the `community` property we have assigned to these trolls. Then run, the algorithm adds to a `community` property to all remaining nodes and updates the property values so that they eventually specify the communities the nodes have been assigned to by the algorithm. The value of the `community` property does not have inherent meaning apart from categorical separation of the detected communities.

```
CALL algo.labelPropagation(
  "MATCH (t:Troll) RETURN id(t) AS id",
  "MATCH (r1:Troll)-[:POSTED]->(t:Tweet)
            <-[:RETWEETED]-(:Tweet)<-[:POSTED]-(r2:Troll)
   RETURN id(r2) AS source,
          id(r1) AS target, count(t) AS weight",
  "OUTGOING",
  {graph:"cypher", write: true,
   partitionProperty: "community", iterations: 200}
)
```

We can then see which Trolls were assigned to each community:

```
MATCH (t:Troll) WHERE exists(t.community)
```

```
RETURN collect(t.screen_name) AS members, t.community
  ORDER BY size(members) DESC LIMIT 10
```

Finally, we can see if there are certain themes that each community was focused on, by inspecting the top-10 common hashtags used by the top-10 largest communities:

```
MATCH (t:Troll) WHERE exists(t.partition)
WITH collect(t) AS members, t.community
  ORDER BY size(members) DESC LIMIT 10
UNWIND members AS t
MATCH (t)-[:POSTED]->(tw:Tweet)-[:HAS_TAG]->(ht:Hashtag)
WITH community, ht.tag AS tag, count(tw) AS num
  ORDER BY community, num DESC
RETURN community, collect(tag)[..10] AS toptags
```

Upon aggregating the hashtags that each community was using, we can see that the group around top influential troll @TEN_GOP was tweeting mainly about right-wing politics (#VoterFraud, #TrumpTrain); the group around @DanaGeezus was more left leaning, but not necessarily positively (#ObamasWishlist, #RejectedDebateTopics); and the group around @gloed_up covered topics in the Black Lives Matter community (#BlackLivesMatter, #Racism, #BLM). Each of these three clusters tended to have a small number of original content generators, with the bulk of the community amplifying the message. For example, one account @TheFoundingSon sent more than 3200 original tweets, averaging about 7 tweets per day. On the other hand, accounts like @AmelieBaldwin authored only 21 original tweets out of more than 9000 sent.

4.3 Visualization

Including the results of graph algorithms in visualization allows us to interpret the results of graph algorithms that otherwise might be difficult to make sense of. Fig. 3 visualizes the three distinct communities identified by our community detection algorithm and the most influential troll accounts within each community, as determined by PageRank. The communities are shown in different colors, node sizes are styled proportionally to their PageRank score, and relationship thickness is styled proportionally to relationship weights, in this case the number of times a troll retweeted another.

5 Evaluation

To demonstrate the performance of Neo4j graph algorithm procedure library we conducted a series of experiments. We ran PageRank, Union Find (connected components), Label Propagation, and Strongly-Connected Components on a number of standard graph datasets available on SNAP [LK14].

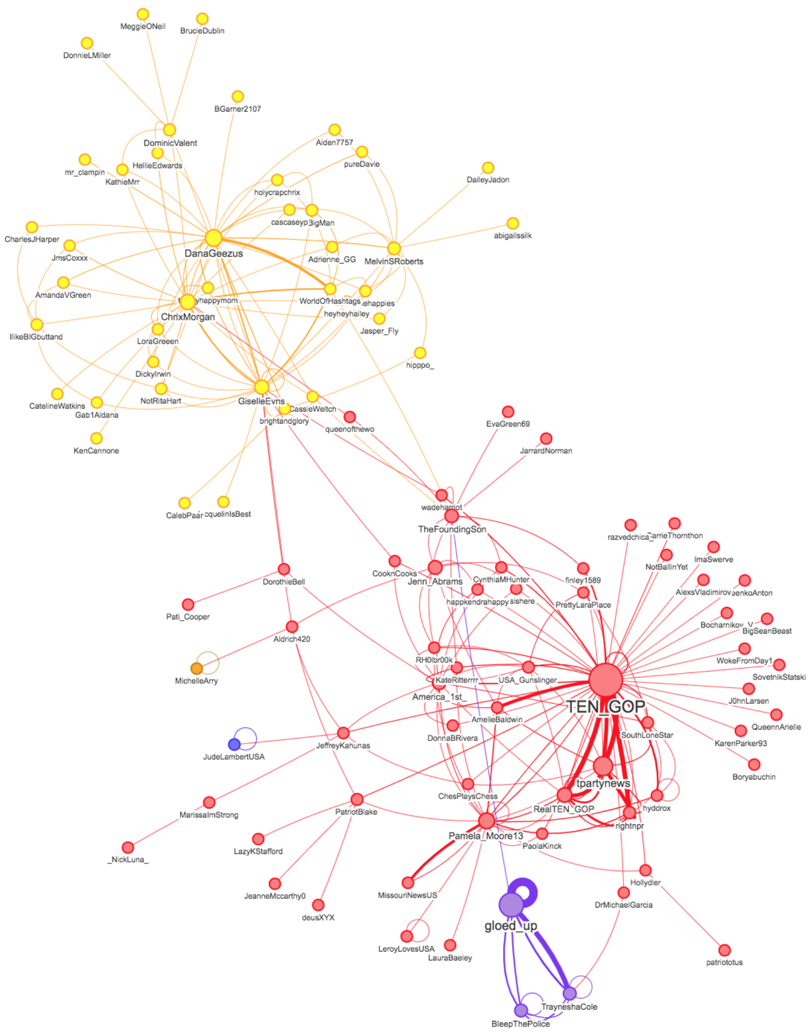


Fig. 3: Visualization of the three distinct communities and the most influential troll accounts within each community.

Setup: We used a machine with four Intel Xeon E7-8870 v3 CPUs, each featuring eighteen cores at 2.1 GHz clock speed, hyper threading, and 45 MB of cache. Additionally, the machine was equipped with 1 TB of DDR4 memory and 1.8 TB of PCIE-SSD storage, in which the Neo4j database files resided. The software stack consisted of Ubuntu Server 16.04 with kernel version 4.4.0, Java(TM) SE 1.8.0_121-b13 for compiling, Java HotSpot(TM) 64-Bit Server VM 25.121-b13 for running, Neo4j Enterprise 3.1.4. Neo4j was configured to a page cache size of 5 GB and the Java VM was allowed to allocate 32 GB of heap while executing Neo4j.

Graph dataset: We have not used the Twitter troll dataset for our experiments, since it is not very large in size. Instead, we used in total eight datasets of various sizes as shown in Tab. 2. Note that the disk size shows the size of the graphs in the Neo4j database, which resides on SSD and gets page cached in-memory. When loaded as graph of interest into the in-memory representation, each graph fitted into the memory the JVM was allowed to allocate.

Graph		#Nodes [M]	#Relationships [M]	Avg. out degree	Disk size [GB]
Pokec	(PK)	1.63	30.62	18.75	0.99
cit-patents	(CP)	3.77	16.52	4.38	0.58
Graphs500-23	(G5)	4.61	129.33	28.05	4.17
soc-LifeJournal1	(LJ)	4.85	68.99	14.23	2.27
DBPedia	(DP)	11.47	116.60	10.16	3.87
Twitter-2010	(TW)	41.65	1468.37	35.25	47.60
Friendster	(FR)	65.61	1806.07	27.53	58.94

Tab. 2: Graph datasets used in measurements.

Algorithms: We ran four algorithms, precisely: PageRank, Union Find, Label Propagation, and Strongly-Connected Components (SCC) on each of the eight graph datasets. The corresponding procedure calls are listed in Tab. 3. As can be seen, we did not set parameters for graph projection, so that the algorithms ran on the complete graph dataset. All procedures were set to write back their results into the graph dataset (`write:true`).

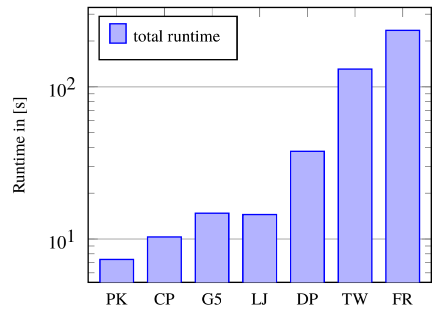
Algorithm	Execution
Pagerank	`CALL algo.pageRank(null, null, {write:true, iterations:20});`
Union Find	`CALL algo.unionFind(null, null, {write:true});`
Label Propagation	`CALL algo.labelPropagation(null, null, 'OUTGOING', {write:true});`
SCC	`CALL algo.scc.iterative(null, null, {write:true});`

Tab. 3: Procedure calls used in measurements.

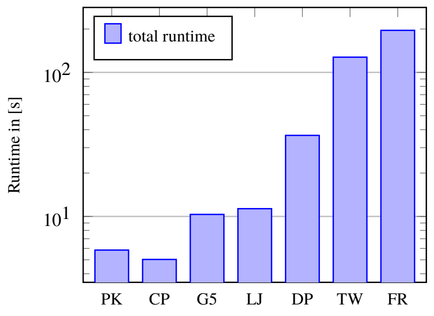
Results: We measured the total runtime of the complete procedure call, which included loading the graph, computing the analysis, and writing back the result. Fig. 4 shows the runtime for each procedure call on each graph dataset. All four algorithms completed within a few seconds for graphs upto 30 M relationships (PK, and CP) and within half a minute for graph upto 120 M relationships (G5, LJ, DP). On the very large graphs with more than 1 G relationships (TW and FR), all procedures finished within a few minutes.

To take a closer look, we also measured the individual runtimes of loading the graph (load), computing the analysis (compute), and writing back the result (write). Fig. 5 reports for each procedure call on each graph dataset the proportion of runtime spent on load, compute, and write. As can be seen most procedure call spent the majority of their runtime in loading the graph into to the representation used by the algorithms. This is as expected, since the whole graph of interest has to be read during load. Although it takes most of the runtime, the loading is still very efficient. Looking at the very large graphs TW and FR, the procedure accomplished to load the graph of interest at a rate of put to 20 000 relationships/s.

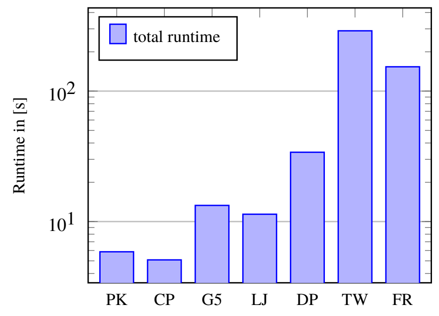
The benefit of loading is revealed during the compute phase. The succinct in-memory representation of the graph of interest facilitates very efficient graph algorithm computation. For instance, PageRank requires to read over all relationships ones per iteration. In



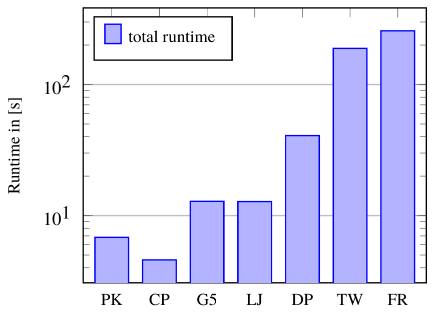
(a) Pagerank.



(b) Union Find.

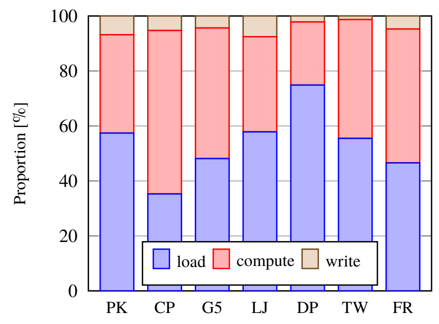


(c) Label Propagation.

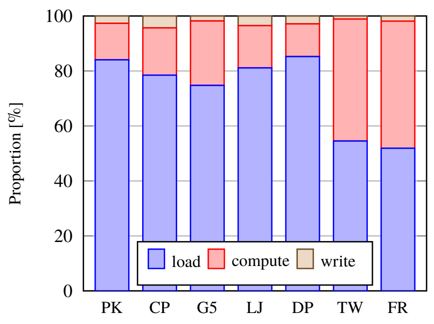


(d) Strongly-Connected Components.

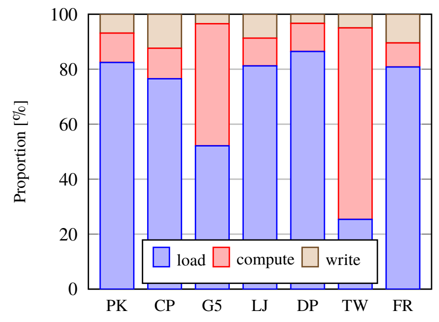
Fig. 4: Total runtimes.



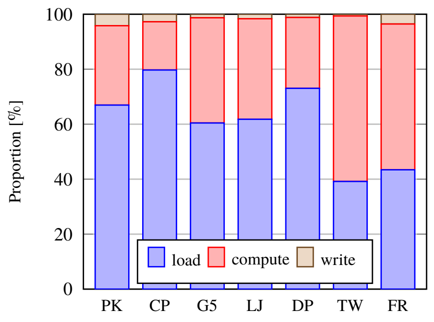
(a) Pagerank.



(b) Union Find.



(c) Label Propagation.



(d) Strongly-Connected Components.

Fig. 5: Proportion of runtime spent on loading, computing, and writing back.

our experiments PageRank was computed with an average over all graph datasets of 275 000 relationships/s and on the Twitter graph with upto 519 000 relationships/s.

6 Related Work

Any system for graph analytics provides an abstract construct for a user to execute analytical graph algorithms without caring about the technical details of an efficient parallel execution. So far, one of the most prominent and common abstractions for graph analytics has been the *vertex-centric* programming model [Lo10, Ma10, SBC10]. A vertex-centric program requires users to provide a compute function that defines the actual analytical computation. A declarative variant worth mentioning is the GSQL [De18].

Since many graph algorithms are based on traversals, the traversal itself provides another common abstraction for graph analytics. *Traversal-based* languages provide native support for graph traversals as part of a domain-specific language, examples are Gremlin [Ro15], GreenMarl [Ho14], GEM [Ru13], and GraphScript [Pa17].

Instead of an dedicated abstraction, declarative graph query language [ARV18, Fr18, Re16, PPvR18] can be extended for analytical purposes by means of composition, grouping and aggregation [An18, Vo17]. Under certain constraints aggregation can be even allowed within recursion [SGL15].

All these approaches provide a means to implement or express algorithms as opposed to the specific graph algorithms themselves. While this is very useful for users developing their own custom algorithms, it unnecessarily complicates matters for users that simply want to deploy well established and understood algorithms. Such users are better served with algorithm libraries.

A very prominent graph algorithms library are the Boost Graph Library[8] [SLL02] and Parallel Boost Graph Library[9]. Both libraries still require code from users in order to extract and feed their graph into the chosen algorithm or input results of one algorithm into another. While Neo4j graph algorithms provide a similar set of algorithms, they are, in addition, well-integration with the graph platform. Users can directly call graph algorithms on a graph database and have their result written back to the graph database, without any additional imperative programming needed. User can also directly leverage the Cypher query capabilities to declaratively select the graph of interest. This is a relevant feature, since the graph database contains in practice significantly more data or data in different shape than the graph of interest, as illustrated in Section 4.2.

For a more comprehensive overview on systems for big graph analytics we refer the reader to [PV17].

[8] `https://www.boost.org/doc/libs/1_66_0/libs/graph/doc/`

[9] `https://www.boost.org/doc/libs/1_66_0/libs/graph_parallel/doc/html/`

7 Conclusion

With a relationship-centered approach one does not look at individuals in isolation, builds understanding of the world by looking at how person, events, and things are connected. For many aspects, the relationships of interest are hidden behind layers of many other connections. For uncovering such hidden relationship, graph data and graph analytics becomes increasingly instrumental. The Neo4j graph platform provides a comprehensive software stack to perform graph analytics, from ingesting the data all the way to visualizing the results. A central piece provided by the graph platform and used in such a graph analytics are the graph algorithms, which we presented in this paper. The graph algorithms provide a comprehensive library of user-defined procedures offering graph analytical algorithm well-integrated with the other components of the graph platform. The library covers algorithms for centrality measuring, path finding, and community detection. All procedures can be called within a Cypher query. Cypher can be also used to project out the graph of interest for called procedure. The procedures can write back the result to the base data or stream it to the calling Cypher query for post processing. The graph algorithms are designed to be executed on large to very large graphs and were tested with several billion nodes and tens of billions of relationships, exhibiting linear performance gain and scalability on large multi-core machines.

In the future, we will continue to apply new research from graph analytics and machine learning. For instance, we are already in the processes of adding performant means for similarity computations including a number of different common similarity measure, such as Jaccard distance. We also look into the generation of k-nearest-neighbour networks. Another considered extension of the library will provide an infrastructure for user-defined algorithms that allows for efficient execution. We also intend to explore new execution models like distributed processing or GPU based approaches.

References

[Ah95] Ahamad, Mustaque; Neiger, Gil; Burns, James E.; Kohli, Prince; Hutto, Phillip W.: Causal Memory: Definitions, Implementation, and Programming. Distributed Computing, 9(1):37–49, March 1995.

[An18] Angles, Renzo; Arenas, Marcelo; Barceló, Pablo; Boncz, Peter A.; Fletcher, George H. L.; Gutierrez, Claudio; Lindaaker, Tobias; Paradies, Marcus; Plantikow, Stefan; Sequeda, Juan; van Rest, Oskar; Voigt, Hannes: G-CORE: A Core for Future Graph Query Languages. In: SIGMOD. ACM, 2018.

[ARV18] Angles, Renzo; Reutter, Juan; Voigt, Hannes: Graph Query Languages. In: Encyclopedia of Big Data Technologies. Springer, 2018.

[Ba50] Bavelas, Alex: Communication Patterns in Task-Oriented Groups. The Journal of the Acoustical Society of America, 22(6):725–730, November 1950.

[Bl08] Blondel, Vincent D.; Guillaume, Jean-Loup; Lambiotte, Renaud; Lefebvre, Etienne: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment, 2008(10):P10008+12, October 2008.

[BP07] Brandes, Ulrik; Pich, Christian: Centrality Estimation in Large Networks. International Journal of Bifurcation and Chaos, 17(7):2303–2318, 2007.

[Br01] Brandes, Ulrik: A Faster Algorithm for Betweenness Centrality. Journal of Mathematical Sociology, 25(2):163–177, 2001.

[Bu09] Buluç, Aydin; Fineman, Jeremy T.; Frigo, Matteo; Gilbert, John R.; Leiserson, Charles E.: Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In: SPAA. ACM, pp. 233–244, 2009.

[CC11] Chu, Shumo; Cheng, James: Triangle Listing in Massive Networks and Its Applications. In: KDD. ACM, pp. 672–680, 2011.

[Da06] Dangalchev, Chavdar: Residual closeness in networks. Physica A: Statistical Mechanics and its Applications, 365(2):556–564, June 2006.

[De18] Deutsch, Alin: Querying Graph Databases with the GSQL Query Language. In: SBBD. SBC, p. 313, 2018.

[Di59] Dijkstra, Edsger W.: A Note on Two Problems in Connexion with Graphs. Numerische mathematik, 1(1):269–271, 1959.

[FI14] Fujiwara, Yasuhiro; Irie, Go: Efficient Label Propagation. In: ICML. JMLR.org, pp. 784–792, 2014.

[Fl63] Flament, Claude: Application of Graph Theory to Group Structure. Prentice-Hall, chapter 3: Balancing Processes, 1963.

[Fr77] Freeman, Linton C.: A Set of Measures of Centrality Based on Betweenness. Sociometry, 40(1), March 1977.

[Fr18] Francis, Nadime; Green, Alastair; Guagliardo, Paolo; Libkin, Leonid; Lindaaker, Tobias; Marsault, Victor; Plantikow, Stefan; Rydberg, Mats; Selmer, Petra; Taylor, Andrés: Cypher: An Evolving Query Language for Property Graphs. In: SIGMOD. ACM, 2018.

[GZB04] Gleich, David F.; Zhukov, Leonid; Berkhin, Pavel: Fast Parallel PageRank: A Linear System Approach. Technical report, Yahoo, 2004.

[HA90] Hutto, Phillip W.; Ahamad, Mustaque: Slow Memory: Weakening Consistency to Enchance Concurrency in Distributed Shared Memories. In: ICDCS. IEEE Computer Society, pp. 302–309, 1990.

[He58] Heider, Fritz: The Psychology of Interpersonal Relations. John Wiley & Sons, 1958.

[HNR68] Hart, Peter E.; Nilsson, Nils J.; Raphael, Bertram: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Trans. Systems Science and Cybernetics, 4(2):100–107, July 1968.

[Ho14] Hong, Sungpack; Salihoglu, Semih; Widom, Jennifer; Olukotun, Kunle: Simplifying Scalable Graph Processing with a Domain-Specific Language. In: CGO. ACM, pp. 208–218, 2014.

[Ko13] Kourtellis, Nicolas; Alahakoon, Tharaka; Simha, Ramanuja; Iamnitchi, Adriana; Tripathi, Rahul: Identifying High Betweenness Centrality Nodes in Large Social Network. Journal of Social Network Analysis & Mining, 3(4):899–914, December 2013.

[KS91] Kumar, Vipin; Singh, Vineet: Scalability of Parallel Algorithms for the All-Pairs Shortest-Path Problem. Journal of Parallel and Distributed Computing, 13(2):124–138, October 1991.

[LK14] Leskovec, Jure; Krevl, Andrej: , SNAP Datasets: Stanford Large Network Dataset Collection, June 2014.

[Lo10] Low, Yucheng; Gonzalez, Joseph; Kyrola, Aapo; Bickson, Danny; Guestrin, Carlos; Hellerstein, Joseph M.: GraphLab: A New Framework For Parallel Machine Learning. In: UAI. pp. 340–349, 2010.

[Ma07] Madduri, Kamesh; Bader, David A.; Berry, Jonathan W.; Crobak, Joseph R.: An Experimental Study of A Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances. In: ALENEX. SIAM, 2007.

[Ma10] Malewicz, Grzegorz; Austern, Matthew H.; Bik, Aart J. C.; Dehnert, James C.; Horn, Ilan; Leiser, Naty; Czajkowski, Grzegorz: Pregel: A System for Large-Scale Graph Processing. In: SIGMOD. ACM, pp. 135–146, 2010.

[ML00] Marchiori, Massimo; Latora, Vito: Harmony in the small-world. Physica A: Statistical Mechanics and its Applications, 285(2–3):539–546, October 2000.

[MNS17] Mehlhorn, Kurt; Näher, Stefan; Sanders, Peter: Engineering DFS-Based Graph Algorithms. The Computing Research Repository, abs/1703.10023, March 2017.

[MR05] Manaskasemsak, Bundit; Rungsawang, Arnon: An Efficient Partition-Based Parallel PageRank Algorithm. In: ICPADS. IEEE Computer Society, pp. 257–263, 2005.

[MS03] Meyer, Ulrich; Sanders, Peter: Δ-stepping: a parallelizable shortest path algorithm. Journal of Algorithms, 49(1):114–152, October 2003.

[NMN01] Nesetril, Jaroslav; Milková, Eva; Nesetrilová, Helena: Otakar Boruvka on minimum spanning tree problem Translation of both the 1926 papers, comments, history. Discrete Mathematics, 233(1-3):3–36, April 2001.

[OO14] Ongaro, Diego; Ousterhout, John K.: In Search of an Understandable Consensus Algorithm. In: ATC. USENIX Association, pp. 305–319, 2014.

[Pa17] Paradies, Marcus; Kinder, Cornelia; Bross, Jan; Fischer, Thomas; Kasperovics, Romans; Gildhoff, Hinnerk: GraphScript: implementing complex graph algorithms in SAP HANA. In: Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017. ACM, pp. 13:1–13:4, 2017.

[PPvR18] Paradies, Marcus; Plantikow, Stefan; van Rest, Oskar: Graph Data Management Systems. In: Encyclopedia of Big Data Technologies. Springer, 2018.

[Pr57] Prim, Robert C.: Shortest Connection Networks And Some Generalizations. Bell System Technical Journal, 36(6):1389–1401, November 1957.

[PV17] Paradies, Marcus; Voigt, Hannes: Big Graph Data Analytics on Single Machines – An Overview. Datenbank-Spektrum, 17(2), July 2017.

[RAK07] Raghavan, Usha Nandini; Albert, Réka; Kumara, Soundar: Near linear time algorithm to detect community structures in large-scale networks. Physical Review E, 76(3):036106–1–11, September 2007.

[Re16] van Rest, Oskar; Hong, Sungpack; Kim, Jinha; Meng, Xuming; Chafi, Hassan: PGQL: a property graph query language. In: GRADES. ACM, p. 7, 2016.

[RN10] Rodriguez, Marko A.; Neubauer, Peter: Constructions from Dots and Lines. Bulletin of the American Society for Information Science and Technology, 36(6):35–41, August 2010.

[Ro15] Rodriguez, Marko A.: The Gremlin Graph Traversal Machine and Language. In: DBPL. ACM, 2015.

[Ru13] Rudolf, Michael; Paradies, Marcus; Bornhövd, Christof; Lehner, Wolfgang: The Graph Story of the SAP HANA Database. In: BTW. volume 214. GI, pp. 403–420, 2013.

[SBC10] Stutz, Philip; Bernstein, Abraham; Cohen, William W.: Signal/Collect: Graph Algorithms for the (Semantic) Web. In: ISWC. Springer, pp. 764–780, 2010.

[SGL15] Seo, Jiwon; Guo, Stephen; Lam, Monica S.: SociaLite: An Efficient Graph Query Language Based on Datalog. IEEE Transactions on Knowledge and Data Engineering, 27(7):1824–1837, 2015.

[SLL02] Siek, Jeremy G.; Lee, Lie-Quan; Lumsdaine, Andrew: The Boost Graph Library - User Guide and Reference Manual. Pearson / Prentice Hall, 2002.

[SRM14] Slota, George M.; Rajamanickam, Sivasankaran; Madduri, Kamesh: BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems. In: IPDPS. IEEE Computer Society, pp. 550–559, 2014.

[Ta72] Tarjan, Robert Endre: Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing, 1(2):146–160, June 1972.

[Ts08] Tsourakakis, Charalampos E.: Fast Counting of Triangles in Large Real Networks without Counting: Algorithms and Laws. In: ICDM. IEEE Computer Society, pp. 608–617, 2008.

[Vo17] Voigt, Hannes: Declarative Multidimensional Graph Queries. volume 280. Springer, pp. 1–37, 2017.

[Wi14] Wickramaarachchi, Charith; Frîncu, Marc; Small, Patrick; Prasanna, Viktor K.: Fast parallel algorithm for unfolding of communities in large graphs. In: HPEC. IEEE, pp. 1–6, 2014.

[Ye70] Yen, Jin Y.: An algorithm for finding shortest routes from all source nodes to a given destination in general networks. Quarterly of Applied Mathematics, 27(4):526–530, January 1970.

[Ye71] Yen, Jin Y.: Finding the *K* shortest loopless paths in a network. Management Science, 17(11):712–716, 1971.