Computer Science Review 62 (2026) 100981

Contents lists available at ScienceDirect

Computer Science Review

journal homepage: www.elsevier.com/locate/cosrev

Data analytics algorithms in property graph databases: A survey

Francesco Cambria*, Francesco Invernici, Anna Bernasconi, Stefano Ceri

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy

ARTICLE INFO

Keywords:
Graph data analytics algorithms
Graph databases
Property graph databases
Performance evaluation of property graph applications

ABSTRACT

As data continues to grow in scale, data analytics algorithms play a crucial role in extracting and refining valuable knowledge from data. At the same time, graph databases are essential for efficiently managing network data; among them, the most popular systems are converging towards property graph databases, thereby adding rich semantics to graph modeling.

In this survey, we explore how data analytics algorithms are supported by property graph databases. First, we provide a comprehensive description of the main graph data analytics algorithms, by classifying and then explaining forty-five algorithms. Then, we examine the twenty most popular graph databases, based on an externally provided usage ranking, and map the data analytics algorithms to them, discovering that only ten property graph databases support some data analytics algorithms. The outcome of this work provides an indication of the coverage of graph data analytics by each property graph database.

Finally, to pragmatically guide potential users in choosing the best available solutions for graph data analytics, we evaluate the performance of three property graph databases (Neo4j, Memgraph, and TigerGraph) selected on the basis of usage ranking, data analytics coverage and availability of an open-source version. Our performance evaluation applies to synthetic datasets of different topologies and increasing sizes, and to five real-world graphs that exhibit different network features.

1. Introduction

In the last two decades, thanks to the exponential growth of large, interconnected data, graph databases have emerged for their effective management of graph data, overcoming the limitations of mainstream databases (e.g., relational) that do not focus on the network structure [1]. Network data naturally capture a variety of domains, as shown in general by Barabasi [2] and specifically in domains such as social analytics [3], biology [4–6], finance [7] and law [8,9].

Along with the emergence of these applications, a new generation of management systems for graph databases (Graph DBMSs) has emerged; [10] lists them in order of "popularity". Graph databases are optimized for storing and querying graph data, often supporting ACID transactions and large-scale datasets [11]. The top-ranked Graph DBMSs support the property graph (PG) model, a rich data model allowing nodes and edges to carry properties and labels [12], which is emerging as a unifying standard for providing schema-level descriptions to graph databases. Coherently, our survey is focused on Graph DBMSs supporting the PG model.

While Data Science is a term that can be traced back to the 1970s [13], it has recently gained attention and popularity, partly thanks to the big data revolution [14,15]. Generally, data science can be defined as the intersection of different and broad interdisciplinary fields, from statistics and computer science to management and sociology, aiming to capture deeper knowledge from data [15,16]. Data science pipelines include a process starting from data collection and preparation (possibly including data enrichment and imputation), to data quality verification, to the application of statistical algorithms and/or machine learning models, culminating in the clear presentation of results [15]. Graph Data Science refers to the specialized set of algorithms and models designed to analyze graphs, leveraging their structure to extract meaningful insights [17].

Data Science methods typically include data analytics methods, which use top-down designed algorithms, and machine learning methods, which use bottom-up, inductive methods. In this survey, we focus on data analytics, as in most graph databases these methods are supported by specialized algorithms. We instead exclude all machine learning methods from our survey; we motivate our choice because, in

* Corresponding author.
Email addresses: francescoluciano.cambria@polimi.it (F. Cambria), francesco.invernici@polimi.it (F. Invernici), anna.bernasconi@polimi.it (A. Bernasconi), stefano.ceri@polimi.it (S. Ceri).

https://doi.org/10.1016/j.cosrev.2026.100981
Received 20 October 2025; Received in revised form 3 April 2026; Accepted 3 April 2026
Available online 13 April 2026

most graph databases, machine learning tasks are mapped to generic pipelines, where the machine learning method is imported from an external library, without a specific connection to the database engine. Some algorithms discussed here are frequently components of machine learning pipelines; however, we analyze them as standalone analytical tools because DBMSs implement them natively for direct analysis, rather than as part of an inductive learning framework. We also exclude from our survey all graph learning methods, because they are also supported by generic pipelines, typically applied to vector-based representations of graphs, without any emphasis on the method. Finally, we also exclude specialized tools designed for in-memory computation and analysis of graph structures [18–20], focusing on in-memory data processing regardless of specific data management system contexts.

Contributions. In this survey, we explore the solutions offered by property graph Database Management Systems for implementing graph data analytics algorithms. The survey addresses key questions, such as: *Which algorithms are supported by each system? How are they encoded? To which scenarios can they be applied?* This review serves as a road map, enabling readers to select the algorithms and the property graph database systems that are best suited to their specific needs. A noticeable aspect of this survey is the comparative performance evaluation of three database engines, which have an open-source version, are chosen among the most widely used engines according to the listing of DB-Engines [10], and at the same time cover most extensively the set of reviewed graph algorithms. Our comparative analyses are applied to synthetic data as well as real-life applications. All these aspects help readers identify the most suitable solution for their needs, considering supported algorithms, the system's accessibility, and performance.

Related Surveys. Many surveys capture the evolution of network analysis; they investigate classes of algorithms designed for network data, highlighting both their capabilities and challenges [21–23]. Other surveys focus on the perspective of specific graph application domains; Borgatti et al. [24] review how network theory is applied to represent social dynamics; Scott [25] reviews how to perform network analysis for the social domain. Other surveys specialize in specific kinds of networks/applications: biological networks [26–28], graph-based fraud detection systems [29–31], and transportation networks [32,33]. Other surveys, instead, study the computational complexities of graph data analytics methods. For instance, the articles [34–36] consider the most common network analysis algorithms, while Jiang [37] focuses on pattern matching.

With respect to graph databases, several surveys have been exploring their various facets. Angles and Gutierrez [38,39] propose a comparison of the different data models for graph databases. Angles et al. [40] also present the foundational features of graph database querying languages, and Kondylakis et al. [41] present recent advancements made by property graph standards. Bonifati et al. investigate how to generate new graph databases [42] and how to query graph databases [43]. Finally, Lopez et al. [44] propose a categorization of graph databases according to data storage type and the data model, and Besta et al. [45] present a complete taxonomy of graph databases. Our survey aims to further expand these analyses, providing an in-depth investigation of data analytics on graphs.

Outline. The survey organization is shown in Fig. 1, which also provides a graphical introduction to the dependencies between the various parts of our survey. In Section 2, we summarize some basic notions about graph data models (including property graphs) and some fundamental network features. Section 3 provides a categorization and description of the algorithms for graph data analytics. In Section 4, we review the most popular graph data management systems, with a strong focus on property graph database management systems. In Section 5, we evaluate the performance of different Graph DBMSs (supporting the property graph data model) and graph data analytics algorithms on synthetic graphs (considering Random, Scale-Free, and Small-World generators) and several cases of real-world datasets. Section 6 concludes this survey.

2. Property graphs and other graph models

A graph can be formally defined as a triple consisting of a set of nodes $N(G)$, a set of edges $E(G)$, also called relationships, and a function that assigns each edge to a pair of nodes, potentially the same node, serving as its endpoints [46]. When the nodes assigned to each edge are ordered, the graph is directed, meaning that every edge has a designated starting and ending node.

In a graph database, the data model is inherently graph-based. This means that both the schema and data instances are represented as graphs, and queries are designed to perform graph-specific operations. Many graph databases adopt the property graph (PG) data model [12], representing data as a directed graph. In this model, both nodes and edges are labeled; labels are unique identifiers within the graph, providing the node's identity. Nodes and edges can have associated ⟨property, value⟩ pairs. Although property graph databases lack an official standard query language, the Graph Query Language (GQL, [47,48]) is emerging

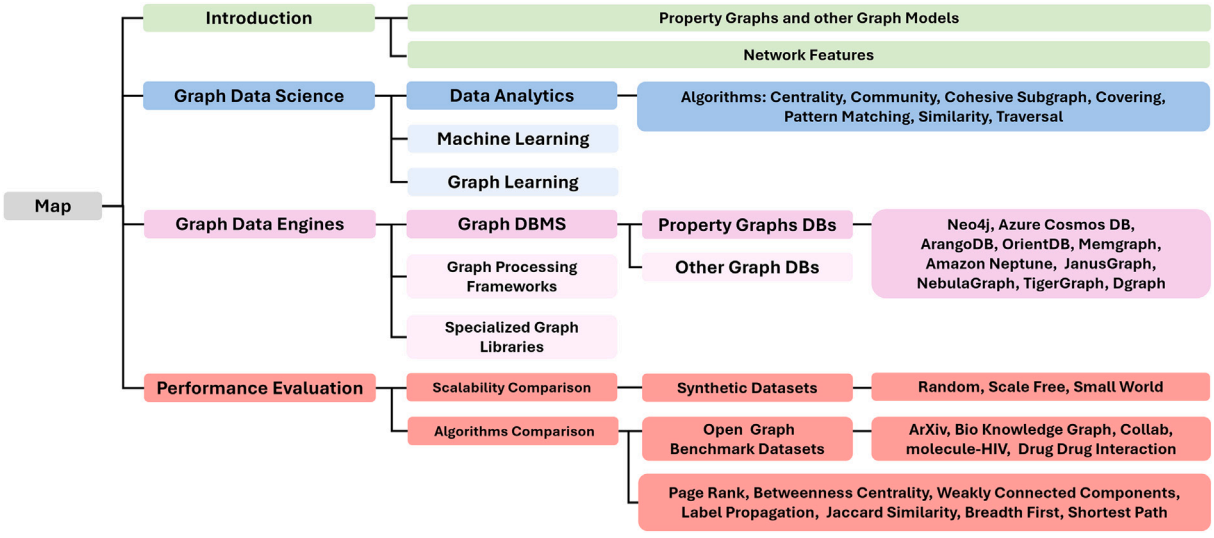

Fig. 1. Outline of the survey.

as a graph-specific complement to SQL for efficient graph navigation. GQL is very similar to Cypher, the most commonly used query language (adopted by Neo4j, the most widely adopted property graph database).

Due to their growing popularity and flexibility in modeling complex scenarios, significant efforts have been made in recent years to enhance and standardize property graph databases. Angles et al. [49] introduce PG-Schema, a framework for defining schema definitions, e.g., by adding generic schema types. Note that when all the relationships in the database have the same type, the relationship identifier is given by the pair ⟨source-node-id, destination-node-id⟩; when instead multiple relationship types are supported, the relationship identifier is given by the triple ⟨relationship-type, source-node-id, destination-node-id⟩.

In addition to identifiers, the PG-keys standard proposal [50] defines a generic mechanism to provide property-based identification to nodes and relationships to prevent duplicates and ensure data integrity. [51–53] introduce PG-triggers, serving as reactive components for monitoring and managing data, as a reaction to the creation or deletion of nodes and edges, as well as the updates of their labels and properties. Significant efforts have been made not only in data management but also in data analytics and mining. For instance, in [54], the *mine graph rule* is introduced as a GQL-based operator for defining and extracting association rules from property graph databases.

In addition to the property graph model, other graph data models offer different approaches to organizing and querying graph data. One such model is the Resource Description Framework (RDF, [55]) model, which represents data as triples (subject-predicate-object) and is widely used in semantic web technologies. Another alternative is the hypergraph model [56], which generalizes the graph structure by allowing edges to connect more than two nodes, making it suitable for more complex relationships. These models, each with their own strengths, provide diverse ways to capture and analyze graph-based data depending on the specific requirements of the application.

2.1. Network features

Graphs can be characterized by various network features that describe their structure, complexity, and impact on graph data science. Qualitatively, graphs can be directed or undirected; the former kind provides a specific direction to each edge (i.e., it assigns to the nodes of an edge respectively the roles of source and target of the edge); note that it may be possible to query the graph by traversing edges from their target node to their source node. Graphs can also be homogeneous or heterogeneous: a homogeneous graph has a single node type and a single edge type, while a heterogeneous graph can contain multiple node and relationship types. From the point of view of their topology, graphs can either be fully connected, meaning every node is reachable from any other node through a path of arbitrary length, or they can be divided into isolated components, i.e., fully connected subgraphs where nodes from a subgraph are not connected to nodes of a different subgraph.

Quantitatively, the size of a graph significantly affects the computational resources required for processing data analysis algorithms. Size is defined by counting either the number of nodes or the number of edges, each influencing the performance of algorithms in distinct ways. For example, the performance of traversal algorithms depends on the number of edges, whereas the performance of clustering algorithms depends on the number of nodes. Closely related to graph size, graph density is defined as the ratio of existing edges to the total possible edges between nodes.

Beyond the total number of edges and their density, another key characteristic is how edges are distributed across the network, which is captured by the statistical distribution of edges between nodes; the most interesting cases include uniform, random [57], and scale-free distributions [2]. Real-world graphs often follow a scale-free distribution, reflecting the preferential attachment principle: nodes tend to connect to other nodes that already have many connections.

The clustering coefficient measures the tendency of nodes to form tightly connected groups; high values indicate strong connectivity, while low values indicate loose connectivity, typical in random graphs. Lastly, another distinctive feature of graphs is the diameter. To explain it, we first define the distance between two nodes as the number of edges in the shortest path connecting them. A node's eccentricity is the maximum distance from that node to any other node in the graph. The graph's diameter is then the largest eccentricity among all nodes, representing the longest shortest path within the graph. Other types of networks are defined by specific value ranges of these features. For example, networks with a high clustering coefficient and a low diameter are known as Small-World graphs [58].

3. Graph data analytics algorithms

Graph data analytics algorithms generally cover a wide range of data analytics tasks, aimed at extracting knowledge from graphs; taking as reference the taxonomies found in [34,35,59], each algorithm can be classified considering the expected result. The resulting categories are:

- **Centrality**, to identify the most important nodes within networks;
- **Community**, to identify clusters in graphs;
- **Cohesiveness**, to identify subgraphs whose nodes are densely connected;
- **Covering**, to identify sets of nodes or edges that satisfy specific constraints;
- **Pattern Matching**, to identify subgraphs that are isomorphic to a given pattern;
- **Similarity**, to identify similar nodes;
- **Traversals**, to identify paths in graphs.

In Table 1 the full list of the algorithms is presented. The algorithms are grouped into classes. For each algorithm, the table lists the mandatory and optional property graph elements required, as well as the produced output.

Table 2 outlines the mathematical characteristics of each algorithm, including time and space complexity at the asymptotic level and whether they are iterative or randomized; note that these figures reflect the general case and may differ in specific contexts.

Selection Criteria. To navigate the vast landscape of network algorithms, we selected those most relevant to the scope of this survey. Our primary criterion was to prioritize methods widely supported by the standard libraries of major graph database vendors. Additionally, we focused on algorithms designed for complex structural analysis, distinguishing them from processes that perform simple transactional lookups.

Differently, we excluded methods that are more aligned with machine learning and graph learning techniques. While powerful, these methods typically require transforming graph data into vectors for external frameworks, rather than executing as native analytical procedures within the database system.

3.1. Centrality

Centrality algorithms are designed to identify the most important nodes within networks [60,61]. These algorithms differ in the specific structural aspects of the graph they investigate and the statistical methods they use to quantify a node's importance [62], represented by a score. Identifying the centrality of the nodes is key to studying a broad spectrum of topics: for instance, discovering super-spreaders in epidemiology [63] or inefficiencies in a transportation network [64].

These algorithms can be broadly categorized into neighbor-based and path-based approaches. Neighbor-based algorithms (Degree Centrality, Eigenvector Centrality, PageRank, and HITS) assess importance based on immediate connections or the recursive influence of neighbors.

In contrast, path-based algorithms (Betweenness Centrality, Closeness Centrality, Harmonic Centrality, and Katz Centrality) analyze the global topology by traversing paths across the network. These

Table 1
Summary of graph data analytics algorithms. Each row corresponds to a specific algorithm divided by category: the first column is the algorithm name; the columns **Node Labels**, **Edge Types**, **Node Properties**, **Edge Properties**, and **Node IDs** represent different property graph elements that are either mandatory (✓), optional (∼) or not used (-); finally the sixth column defines the output format. The output format is expressed as either a new node property; a set of existing nodes (expressed as node identifiers); an existing edge (expressed as an edge identifier); an existing path or tree (a sequence of edges); or a specific new unique feature of the graph (a metric value, a specific node, or a specific edge).

Algorithm	Node Labels	Edge Types	Node Properties	Edge Properties	Node IDs	Output
Centrality						
Degree Centrality	✓	✓	–	∼	–	Node Property
Eigenvector Centrality	✓	✓	–	∼	∼	Node Property
PageRank	✓	✓	–	∼	∼	Node Property
Hyperlink-Induced Topic Search (HITS)	✓	✓	–	–	–	Node Property
Betweenness Centrality	✓	✓	–	∼	–	Node Property
Closeness Centrality	✓	✓	–	–	–	Node Property
Harmonic Centrality	✓	✓	–	–	–	Node Property
Katz Centrality	✓	✓	–	–	–	Node Property
Influence Maximization	✓	✓	–	–	–	Node Property, Nodes
Community						
Weakly Connected Components	✓	✓	∼	∼	–	Node Property
Strongly Connected Components	✓	✓	–	–	–	Node Property
Label Propagation	✓	✓	∼	∼	–	Node Property
Speaker-Listener Label Propagation	✓	✓	∼	∼	–	Node Property
Hop Preference & Node Propagation (HANP)	✓	✓	∼	∼	–	Node Property
Modularity Optimization	✓	✓	∼	∼	–	Node Property
Louvain	✓	✓	∼	∼	–	Node Property
K-means Clustering	✓	–	✓	–	–	Node Property
Maximum k-cut	✓	✓	–	–	–	Node Property
Spectral Clustering	✓	✓	∼	–	–	Node Property
Cohesiveness						
K-Core Decomposition	✓	✓	–	–	–	Node Property
Biconnected Components	✓	✓	–	–	–	Node Property
Covering						
Graph Coloring	✓	✓	–	–	–	Node Property
Maximal Independent Set	✓	✓	–	–	–	Nodes
Maximum Matching	✓	✓	–	–	–	Nodes, Edges
Pattern Matching						
Triangle Counting	✓	✓	–	–	–	Nodes, Value
K-clique	✓	✓	–	–	–	Node Property
Maximum Common Subgraph	✓	✓	–	–	–	Nodes, Edges
Similarity						
Common Neighbors	✓	✓	–	–	✓	Nodes
K-Nearest Neighbors	✓	–	✓	–	✓	Edge
Jaccard Similarity	✓	✓	–	∼	✓	Edge
Overlap Coefficient	✓	✓	–	∼	✓	Edge
Cosine Similarity	✓	✓	–	∼	✓	Edge
Adamic-Adar	✓	✓	–	∼	✓	Edge
Resource Allocation	✓	✓	–	–	✓	Edge
Traversal						
Breadth First Search	✓	✓	–	–	✓	Path
Depth First Search	✓	✓	–	–	✓	Path
Shortest Path	✓	✓	–	∼	✓	Path
All Pairs Shortest Path	✓	✓	–	∼	✓	Path
Yen's Shortest Path	✓	✓	–	∼	✓	Path
A* Shortest Path	✓	✓	–	∼	✓	Path
Graph Edit Distance	✓	✓	–	–	–	Nodes, Edges
Minimum Weight Spanning Tree	✓	✓	–	∼	✓	Tree
Minimum Directed Steiner Tree	✓	✓	–	∼	✓	Tree
Cycle Detection	✓	✓	–	–	–	Path
Maximum Flow Problem	✓	✓	–	✓	–	Path

metrics highlight nodes that act as bridges, rather than just those with high connectivity.

Degree centrality. It measures a node's importance based on the number of its incoming or outgoing relationships [65]. The direction of relationships can either be ignored or considered; in the latter case, only outgoing relationships are used. Additionally, the weight of relationships can be included [66].

Eigenvector centrality. Also called Eigencentrality or Prestige Score, it is based on the idea that the importance of a node depends not only on the relationships with adjacent nodes but also on their value of centrality [67]. Generally, a node with a high Eigenvector Centrality score is connected to many nodes that themselves have high scores. Eigenvector Centrality can introduce a bias toward a specific set of nodes, giving them a higher initial centrality score.

PageRank. Introduced in [68], it was developed to evaluate the importance of web pages by analyzing the network of connections. Similar to Eigenvector Centrality, the core principle of PageRank is that the centrality of a node relies on the centrality of the pages directly connected

Table 2
Summary of mathematical abstractions for graph data analytics algorithms. The time and space complexity of each algorithm, along with their iterative or random nature, are reported. The notation used for complexity analysis is defined as follows: $|N|$ and $|E|$ represent the cardinality of nodes and edges, respectively, while k denotes the number of iterations. Algorithm-specific parameters include R for the number of influence Maximization simulations, c for the number of communities in Speaker-Listener Label Propagation and K-means clustering, and K as the input parameter for K-means clustering, K-nearest neighbors, K-clique, and Yen's Shortest Path. Additionally, v indicates the vector dimensionality for K-means clustering, D is the maximum node degree, p represents the average path length for Yen's Shortest Path, and n denotes the number of target nodes for the Minimum directed Steiner Tree.

Algorithm	Time Complexity	Space Complexity	Iterative	Random
Centrality				
Degree Centrality	$O(\|E\|)$	$O(\|N\|)$	–	–
Eigenvector Centrality	$O(k \cdot \|E\|)$	$O(\|N\| + \|E\|)$	✓	–
PageRank	$O(k \cdot \|E\|)$	$O(\|N\| + \|E\|)$	✓	–
Hyperlink-Induced Topic Search (HITS)	$O(k \cdot \|E\|)$	$O(\|N\| + \|E\|)$	–	–
Betweenness Centrality	$O(\|E\|^2)$	$O(\|N\| + \|E\|)$	–	✓
Closeness Centrality	$O(\|E\|^2)$	$O(\|N\|)$	–	–
Harmonic Centrality	$O(\|E\|^2)$	$O(\|N\|)$	–	–
Katz Centrality	$O(k \cdot \|E\|)$	$O(\|N\| + \|E\|)$	–	–
Influence Maximization	$O(k \cdot \|N\| \cdot R \cdot \|E\|$	$O(\|N\| + \|E\|)$	✓	✓
Community				
Weakly Connected Components	$O(\|N\| + \|E\|)$	$O(\|N\|)$	–	–
Strongly Connected Components	$O(\|N\| + \|E\|)$	$O(\|N\| + \|E\|)$	–	–
Label Propagation	$O(k \cdot \|E\|)$	$O(\|N\|)$	✓	–
Speaker-Listener Label Propagation	$O(k \cdot \|E\|)$	$O(\|N\| \cdot c)$	✓	✓
Hop Preference & Node Propagation (HANP)	$O(k \cdot \|E\|)$	$O(\|N\|)$	✓	✓
Modularity Optimization	$O(\|N\| \log^2 \|N\|)$	$O(\|N\| + \|E\|)$	✓	–
Louvain	$O(\|N\| \log \|N\|)$	$O(\|N\| + \|E\|)$	✓	–
K-means Clustering	$O(k \cdot c \cdot v \cdot \|N\|)$	$O(\|N\| \cdot v + c \cdot v)$	✓	✓
Maximum k-cut	$O(\|N\|^3)$	$O(\|N\| + \|E\|)$	–	–
Spectral Clustering	$O(\|N\|^3)$	$O(\|N\|)$	–	–
Cohesiveness				
K-Core Decomposition	$O(\|N\| + \|E\|)$	$O(\|N\| + \|E\|)$	–	–
Biconnected Components	$O(\|N\| + \|E\|)$	$O(\|N\| + \|E\|)$	–	–
Covering				
Graph Coloring	$O(\|N\|^2 + \|E\|)$	$O(\|N\| + \|E\|)$	✓	–
Maximal Independent Set	$O(\|N\| + \|E\|)$	$O(\|N\|)$	✓	–
Maximum Matching	$O(\|E\| \cdot \sqrt{\|N\|}$	$O(\|N\| + \|E\|)$	✓	–
Pattern Matching				
Triangle Counting	$O(\|E\|^{1.5})$	$O(\|N\| + \|E\|)$	–	–
K-clique	$O(\|N\|^K)$	$O(\|N\| + \|E\| + K \cdot \|N\|)$	–	–
Maximum Common Subgraph	$O(3^{\frac{\|N_1\|+\|N_2\|}{3}})$	$O((\|N_1\| \cdot \|N_2\|)^2)$	–	–
Similarity				
Common Neighbors	$O(\|N\|^2 \cdot D)$	$O(2\|N\| \cdot D)$	–	–
K-Nearest Neighbors	$O(\|N\| \log \|N\|)$	$O(K \cdot \|N\|)$	✓	✓
Jaccard Similarity	$O(\|N\|^2 \cdot D)$	$O(2\|N\| \cdot D)$	–	–
Overlap Coefficient	$O(\|N\|^2 \cdot D)$	$O(2\|N\| \cdot D)$	–	–
Cosine Similarity	$O(\|N\|^2 \cdot D)$	$O(2\|N\| \cdot D)$	–	–
Adamic-Adar	$O(\|N\|^2 \cdot D)$	$O(2\|N\| \cdot D)$	–	–
Resource Allocation	$O(\|N\|^2 \cdot D)$	$O(2\|N\| \cdot D)$	–	–
Traversal				
Breadth First Search	$O(\|N\| + \|E\|)$	$O(\|N\|)$	–	–
Depth First Search	$O(\|N\| + \|E\|)$	$O(\|N\|)$	–	–
Shortest Path	$O(\|E\| \log \|N\|)$	$O(\|N\|)$	–	–
All Pairs Shortest Path	$O(\|N\|^3)$	$O(\|N\|^2)$	–	–
Yen's Shortest Path	$O(K \cdot \|N\|(\|E\| + \|N\| \log \|N\|)$	$O(\|N\| + \|E\| + K \cdot p)$	–	–
A* Shortest Path	$O(\|E\| \log \|N\|)$	$O(\|N\|)$	–	–
Graph Edit Distance	$O(\|N\|!)$	$O(\|N\|!)$	–	–
Minimum Weight Spanning Tree	$O(\|E\| \log \|V\|)$	$O(\|N\|)$	–	–
Minimum Directed Steiner Tree	$O(3^n\|N\| + 2^n\|N\|^2 + \|N\|^3)$	$O(2^n\|N\|)$	✓	–
Cycle Detection	$O(\|N\| + \|E\|)$	$O(\|N\|)$	✓	–
Maximum Flow Problem	$O(\|N\| \cdot \|E\|^2$	$O(\|N\| + \|E\|)$	✓	–

to it. In the original implementation, PageRank was defined as the solution of the equation: $PR_i(n) = (1-d) + d * \sum_{v \in N_{in}} \frac{PR_{i-1}(v)}{N_{out}(v)}$, where $PR_i(n)$ denotes the PageRank score of the node n at step i; d (damping factor) represents the loss of interest in the node (a low-weight connection to a random node in the graph); $v \in N_{in}$ is the set of nodes connected to n by an incoming edge; and $v \in N_{out}$ is the set of nodes connected to n by an outgoing edge. The equation is solved iteratively, to find an approximate solution [69–71].

Hyperlink-induced topic search. HITS [72] measures centrality using two scores: the authority score, which evaluates the importance of a node as "pointed" by other nodes, and the hub score, which evaluates the significance of a node as "pointer" to other nodes; authority and hub scores are computed by two equations, similar to the ones introduced in PageRank, which are solved iteratively by alternating one step for authority and one step for hub. Upon convergence, authority scores are similar to PageRanks, while high hub scores correspond to nodes

that are good "pointers" (e.g., in a network about care providers, nodes pointing to the "highly linked" -best- care providers).

Betweenness centrality. It measures how often a node appears on the shortest paths between other pairs of nodes, providing a useful estimate of how much information flows through that node [73]. Betweenness centrality is calculated in two steps: first, find the shortest paths between all pairs of nodes; then, determine the pair-dependencies, i.e., the ratio of shortest paths between each pair that pass through a given node. Several methods have been proposed focusing on how to solve this challenge. For instance, Brandes et al. [74,75] propose a faster custom algorithm for unweighted graphs achieving an approximate solution through random sampling; for weighted graphs, the Dijkstra algorithms (later evaluated in Section 3.7), are generally used.

Closeness centrality. It is traditionally defined as the inverse of a node's average distance to all other nodes in the graph [76]. Nodes that are closer to all others can efficiently spread information across the entire graph. For each node, the algorithm calculates the sum of its distances to all other nodes, which is then inverted and normalized to obtain the average distance. The resulting formula for Closeness Centrality is: $CC(n) = \frac{N-1}{\sum_{v \in N} d_{nv}}$, where $CC(n)$ denotes the Closeness Centrality score of the node n, N is the set of all the nodes v, and d_{nv} is the length of the shortest path from n to v. A challenge arises when defining distance in graphs that are not fully connected; the Wasserman-Faust formula [77] modifies the original calculation by expressing closeness as the ratio of the fraction of reachable nodes to the average distance.

Harmonic centrality. Introduced in [78], it is a variation of Closeness Centrality designed to handle graphs that are not fully connected. Instead of summing the distances from a node to all others, this approach sums the inverse of those distances: $HC(n) = \sum_{v \in N} \frac{1}{d_{nv}}$, where $HC(n)$ denotes the Harmonic Centrality score of the node n, N is the set of all the nodes v, and d_{nv} is the length of the shortest path from n to v. Unconnected graphs can be handled because terms with infinite distance become zero in the sum.

Katz centrality. Introduced in [79], it addresses some limitations of both Betweenness and Closeness Centrality, which assume that information flows only along the shortest paths [80]. Katz centrality, instead, sums all walks starting from a node, weighting them based on their length. The algorithm is calculated as $KC(n) = \sum_{i=1}^{\infty} p_i(n) * \alpha^i$, where $KC(n)$ denotes the Katz Centrality score of the node n, $p_i(n)$ are the number of paths of increasing length i starting from n, and α is an attenuating factor smaller than 1 that is used to weight the paths according to their length.

Influence maximization. Defined in [81], it aims to find the set of nodes that maximizes the spread of influence across a network. In models that represent the spread of ideas in a graph, a node is typically considered active if it adopts the idea and inactive if it does not. A node's tendency to become active increases as neighboring nodes become active. In [81], the influence of a set of nodes is defined as the expected number of active nodes at the end of a process initiated by setting those nodes as active. The Influence Maximization problem seeks to identify a node set that maximizes the node set's influence. Various approaches have been developed to find solutions; among them, the Greedy algorithm [81] starts with a random set of nodes and incrementally adds to the set those nodes maximizing the influence spread. To improve efficiency, the Cost-Effective Lazy Forward (CELF) algorithm [82] uses a lazy evaluation mechanism to eliminate nodes early in the process, significantly reducing the number of iterations.

3.2. Community

Community Detection algorithms are a class of graph data analytics algorithms designed to identify clusters in a graph, grouping nodes that play similar roles within the network [83,84]. Communities are usually represented as properties or labels attached to each node. Identifying communities in a graph provides valuable insights for various applications: for instance, in social networks for sentiment analysis [85] or to refine user profiling [86].

Most community detection methods can be classified by their optimization strategy: connectivity-based (Weakly and Strongly Connected Components), relying on reachability to group nodes that are structurally linked; propagation-based (Label Propagation, Speaker-Listener Label Propagation, and HANP), using iterative local consensus where nodes adopt the majority label of their neighbors; metric-based (Modularity Optimization and Louvain), focusing on maximizing a specific quality function to ensure high internal density within clusters; and attribute-based (K-means Clustering), grouping nodes based solely on property similarity, completely ignoring structural edges.

Weakly connected components. Described in [87], it identifies communities, or components, as sets of connected nodes in undirected graphs. Sutton et al. [88] propose an efficient algorithm implementation to extract weakly connected components via subgraph sampling.

Strongly connected components. The SCCs algorithm, also described in [87], identifies communities, or components, as sets of connected nodes in directed graphs where every node is reachable from every other node, i.e., there is a directed edge path between them. Classical approaches to finding strongly connected components are [89,90].

Label propagation. It is a bottom-up clustering algorithm inspired by epidemic spreading [91]. It operates on the principle that a node should adopt the most common label among its neighbors. It can optionally use relationship properties as weights to determine influence based on connections and node properties both as weights for neighbor nodes and as a seed for initial labeling. If no initial labels are provided, each node is assigned a unique label, which is then iteratively propagated. At each step, a node updates its label to match the most frequent label among its neighbors. The process continues until every node holds the majority label of its neighbors.

Speaker-listener label propagation. Introduced in [92], it expands Label Propagation so that nodes can belong to multiple overlapping communities. The algorithm is designed to mimic human communication behavior, where each node can act both as a speaker and a listener, while also maintaining a memory to accumulate knowledge. The process begins with each node being assigned a unique label, representing its own initial community. Each node also has a memory to store the labels it receives over time. The algorithm then iterates through two main steps: the *speaking phase*, where each node randomly selects one of the labels stored in its memory and broadcasts it to its neighbors, and the *listening phase*, in which each node receives multiple labels—one from each of its neighbors—and selects the most frequently received label, which it then stores in its memory. After a fixed number of iterations, each node retains the most frequently stored labels.

Hop attenuation and node preference. The HANP algorithm [93] is a variation of Label Propagation designed to enhance community detection. It achieves this by introducing two mechanisms. The first, Hop Attenuation, assigns to each label a score that decreases as it propagates from its origin. This limits label propagation to nearby nodes and prevents labels from spreading too widely across the network. The second mechanism, Node Preference, incorporates node weights based on their degree. When selecting a label to propagate to node i, each label from a neighboring node j is assigned a weight $w_j(L)$, calculated as: $w_j(L) = s_j(L) \cdot (deg_j)^m \cdot w_{ij}$, where $s_j(L)$ is the Hop Attenuation score; $(deg_j)^m$ represents the degree of node j; and m is a parameter that determines whether higher-degree nodes ($m > 0$) or lower-degree nodes ($m < 0$) have more influence. w_{ij} is the weight of the edge between nodes i and j.

Modularity optimization. It is a community detection algorithm that focuses on maximizing modularity [94,95], i.e., the density of connections of nodes within a community [2]. It is calculated as $M = \sum_{i=1}^{c}(e_{ii} - a_i^2)$, where c is the number of communities within the graph; the term e_{ij} is the count of edges that connect nodes in community i to nodes in community j; and the term a_i represents the fraction of all edges that connect to nodes in community i, calculated as $a_i = \sum_j e_{ij}$. Graphs with high modularity scores have many connections between nodes within the same community and few connections between nodes in different communities. The algorithm evaluates the effect of moving a node to any of its neighboring communities. The node is then assigned to the community so that the highest modularity gain is achieved. When the effect of all moves is to decrease modularity, the node remains in its original community.

Louvain. This algorithm, introduced in [96], is an iterative greedy heuristic that identifies a hierarchy of communities by optimizing modularity [97]. Starting from a given or predefined community, the algorithm assigns new communities to nodes based on the principles of Modularity Optimization. After updating all nodes, the algorithm moves to the next phase by collapsing each community into a single meta-node. Edges within a community are summed into a self-loop (i.e., an edge that connects a node to itself) on the meta-node, while edges between communities are merged with weights equal to the total inter-community edge weights. This process is repeated until convergence.

K-means clustering. Used in [98], it identifies c communities within a graph. Unlike other community detection methods, it does not consider relationships but instead relies solely on node properties. The algorithm starts by selecting c random nodes as centroids for the communities. It then calculates the Euclidean distance for all other nodes and assigns each to the nearest centroid. Once assigned, new centroids are computed; this process repeats iteratively until the communities are stable.

Maximum k-cut. It is an algorithm designed to solve the k-cut problem [99]. A k-cut partitions the nodes of a graph into k non-overlapping communities. The maximum k-cut problem seeks the k-cut that maximizes the sum of the weights of relationships connecting nodes from different communities. As finding an exact solution is computationally expensive and time-consuming, most algorithms aim to find an approximation. Among them, [100] employs a greedy adaptive search procedure to efficiently approximate an optimal solution.

Spectral clustering. This algorithm [101] looks for communities in the graph by using the connectivity between nodes. First, it computes the similarity matrix, in which two nodes are connected if their similarity metric is above a threshold; K-Nearest Neighbors [102] is typically used as a similarity metric. Then, the similarity matrix is projected into a lower dimensionality, and in this space, the nodes that are close together are clustered in the same community.

3.3. Cohesiveness

Cohesive subgraphs are subgraphs whose vertices are densely connected [77,103]. After extracting cohesive subgraphs, they can be used for various purposes, such as visual analysis in social networks [104] or keyword extraction from a text graph [105].

K-core decomposition. First introduced in [106], it partitions a graph's nodes into groups based on their degree and the graph's topology. A k-core is the largest subgraph where each node has at least k connections. Lower k-cores include all higher k-cores as subsets. Standard K-Core Decomposition algorithms iteratively remove the node with the lowest degree, along with its edges, reducing the degree of its neighbors by one. This process continues until the graph is empty. Each node is assigned a k-core value equal to its degree just before removal. Various implementations of the algorithm have been developed to improve efficiency and speed through parallelization and multicore processing [107,108].

Biconnected components. These are maximal subgraphs of a graph in which every pair of nodes is connected by two disjoint paths, meaning there is no single node whose removal would disconnect the component. In other words, a biconnected component remains connected even after the removal of any single node. Standard algorithms for finding biconnected components work by identifying articulation points, i.e., nodes whose removal increases the number of connected components within the graph. The algorithm then traverses from these articulation points to other nodes, which together form a biconnected component [109–111].

3.4. Covering

Covering algorithms select a set of nodes or relationships that are optimal under specific constraints. Covering algorithms have various applications, such as finding districts within territories for optimizing the placement of schools [112] or police patrols [113].

Graph coloring. It assigns a color to each node in a graph while optimizing two objectives: 1) ensuring that no two adjacent nodes share the same color and 2) minimizing the total number of colors used [114]. A variation of the algorithm, known as k Coloring, constrains the solution to use exactly k colors. Various approaches have been developed, including iterative optimization methods based on Greedy algorithms [115] or Quantum Annealing [116].

Maximal independent set. This kind of algorithm finds groups of nodes in a graph that are not connected to each other [117]; a graph can have multiple maximal independent sets. These algorithms are typically based on iterative methods, such as Greedy algorithms [118,119].

Maximum matching. In graph theory, a Matching is a selection of edges that pair up nodes so that no node belongs to more than one pair. In other words, each node is connected to at most one other node within the chosen set of edges. Maximum Matching algorithms look for the largest possible matching that contains the maximum number of relationships. One of the algorithms that aims to find the Maximum Matching is the Blossom algorithm [120,121] which is based on the concept of blossoms, i.e., odd-length cycles, to iteratively find the pairs corresponding to the largest matching.

3.5. Pattern matching

Pattern Matching involves finding all subgraphs in a large graph that are isomorphic to a given pattern, where the pattern is defined as a possibly ordered set of nodes and relationships [122]. Graph pattern matching is the fundamental principle behind any graph database query language, as querying a graph involves extracting specific patterns. These patterns can then be applied to various tasks. For example, in [123], a pattern-matching algorithm is used for a movie recommendation system, and in [124], it helps detect spamming activity in large-scale Web graphs. Additionally, Pattern Matching can be extended to the task of finding all instances of a subgraph in a larger graph, but in a corpus of multiple graphs. In this sense, Pattern Matching is also referred to as Graph Matching.

Triangle counting. It refers to those algorithms that extract and count all patterns shaped as triangles, i.e., a pattern of three nodes in which each node is connected to the other two. The count of triangles is also used to compute the Local Clustering Coefficient. Triangle counting algorithms generally fall into two types of approaches: list-based [125,126] and map-based [127,128]. In the list-based approach, the algorithm checks the list of common neighbors for every pair of neighboring nodes. In the map-based approach, the algorithm uses a map to store all neighboring nodes and checks if a node's neighbors appear in the map of

its neighboring nodes. Triangle counting is typically defined for unweighted, undirected graphs, though some exceptions exist for directed graphs, where the direction of interest must be specified [129].

K-clique. These algorithms generalize triangle counting by identifying patterns as sets of k fully connected nodes [130]. For a given node, a maximal clique is defined as the largest clique of size k that includes that node. K-clique algorithms can also be considered a part of cohesive subgraphs, as the resulting subgraphs have maximum density due to all nodes being fully connected.

Maximum common subgraph. The problem of finding the Maximum Common Subgraph (MCS) refers to the task of looking for the largest possible subgraph that is present in a set of multiple, two or more, graphs. Specific variations of the algorithms introduce additional constraints to the problem, such as looking for the subgraph with the largest number of relationships (Maximum Common Edge Subgraph) or requiring that the subgraph has all the relationships between all the nodes it contains (Maximum Common Induced Subgraph). In [131], McGregor proposed one of the first strategies to find the MCS based on backtracking the tree-like bijective mapping of a sample of nodes of the first graph to a sample of nodes of the second. More recently, instead, Cao et al. presented in [132] a backtracking procedure for exact MCS based on subgraph isomorphism techniques.

3.6. Similarity

Similarity algorithms measure the affinity between two nodes in a graph by analyzing either their attributes, such as shared properties or labels, or their network relationships, including common neighbors or connectivity patterns. Since similarity algorithms produce a value between two nodes, this value can generally be interpreted as a new type of relationship within the graph, where the similarity score is stored. Similarity measures have been applied in various domains, such as predicting links between papers in citation networks [133] and recommending connections between users in social networks [134].

Common neighbors. This algorithm measures the similarity between two nodes by counting the number of neighbors they share. The idea is that if two nodes have many common neighbors, they are likely to be similar or connected in some meaningful way. A higher number of common neighbors suggests greater similarity, while zero common neighbors indicates no similarity between the nodes [135].

K-nearest neighbors. K-NN [136,137] calculates the distance between all pairs of nodes in the graph so that for each node it can find the k that are nearest neighbors. Unlike other similarity algorithms, K-NN does not consider relationships but instead relies solely on the properties of the nodes to compute the distance. The initial set of k neighbors for each node is selected randomly and then refined over several iterations. The algorithm stops when the neighbor lists converge. Many implementations have refined this process to achieve faster results. For example, in [138], the algorithm selects potential neighbors based on the assumption that the neighbors-of-neighbors are most likely to be the nearest ones.

Jaccard similarity. Introduced in 1901 by Paul Jaccard [139], it measures the similarity between two nodes by comparing their connected neighbors. The Jaccard Score between nodes A and B is computed as $J = \frac{|A\cap B|}{|A\cup B|} = \frac{|A\cap B|}{|A|+|B|-|A\cap B|}$, where the term $|A \cap B|$ represents the number of nodes connected to both A and B (their Common Neighbors), while $|A \cup B|$ represents the number of nodes connected to either A or B (their Total Neighbors). The score can include a weight for edges.

Overlap coefficient. Also known as the Szymkiewicz–Simpson coefficient, it is a variation of the Jaccard similarity that computes the similarity index as $O = \frac{|A\cap B|}{min(|A|,|B|)}$, where $\min(|A|, |B|)$ is the number of neighbors of the less connected node between A and B. This change results in inherently higher similarity scores. Two nodes can achieve the highest possible score even if their neighbors do not overlap perfectly, as long as the set of neighbors of the node with more connections contains the set of neighbors of the other node. The score can include a weight for edges.

Cosine similarity. It evaluates the similarity between two nodes based on the concept of cosine similarity between vectors. Here, each node is represented as a vector, whose components correspond to the weights of its connecting edges to all other nodes. The Cosine Similarity Score between two nodes A and B is $C = \sum_i \frac{\alpha_i \cdot \beta_i}{\sqrt{\sum_i \alpha_i^2} \cdot \sqrt{\sum_i \beta_i^2}}$, where α_i and β_i are the weights of node i to the other nodes. The score can also be used for unweighted graphs by assigning unitary weights to all edges.

Adamic-adar. First introduced in [140] for link prediction in social networks, it calculates the similarity between two nodes. The Adamic-Adar similarity score for nodes A and B corresponds to $AA = \sum_{n\in N(A)\cap N(B)} \frac{1}{\log |N(n)|}$, where $n \in N(A) \cap N(B)$ represents the set of nodes that are common neighbors of A and B, and $|N(n)|$ is the degree (number of neighbors) of node n. In essence, the Adamic-Adar algorithm measures the similarity between two nodes by summing the inverse of the logarithms of the degrees of their shared neighbors; nodes that share common neighbors with low degrees tend to have higher scores. Murata and Moriyasu [141] suggested a modification of the Adamic-Adar formula to account for weighted relationships.

Resource allocation. This algorithm was first introduced in [142] as a link prediction measure optimized for the physical allocation of resources in graph processing. The Resource Allocation similarity score for nodes A and B is $RA = \sum_{n\in N(A)\cap N(B)} \frac{1}{|N(n)|}$, where $|N(n)|$ is the degree (number of neighbors) of node n. Similarly to the Adamic-Adar score, it reduces the importance of higher-degree nodes when calculating similarity.

3.7. Traversal

Traversal algorithms aim to identify topological structures by traversing the graphs. The output of these algorithms is typically represented as the list of nodes of a structure with all relevant information attached, including the ordered sequence of nodes and edges, and, if applicable, the length or the sum of relationship weights.

Traversal algorithms can be categorized by their objective: exploration, pathfinding, and searching for a specific structure.

Exploration algorithms (Breadth-First Search, and Depth-First Search), focus on visiting nodes in a specific order to analyze reachability.

Pathfinding algorithms, instead, (Shortest Path, All-Pairs Shortest Path, Yen's Shortest Path, and A* Shortest Path, Maximum Flow Problem) are optimization-focused; they seek the most efficient route between specific nodes.

Finally, specialized structure-finding algorithms (Minimum Weight Spanning Tree, Minimum Directed Steiner Tree) aim to connect a set of nodes, constructing a tree-like structure rather than a linear path.

Traversal algorithms help define key network metrics, such as the diameter used in Betweenness Centrality. Traversal algorithms are used in various applications; one of them is VLSI routing [143], i.e., the process of determining the optimal paths for electrical connections between components on a chip.

Breadth-first search. BFS was first introduced in [144]; it explores a graph by starting from a chosen *root* node and visiting all other nodes forming a tree-like structure; the tree is built, at each iteration, by adding to every node all the nodes connected to it. The method starts by enqueuing the root node; while the queue is not empty, it processes each node by storing its information, enqueuing all its unvisited neighbors,

and then dequeuing it. During a search, nodes can be selected, e.g., by computing arbitrary predicates over their properties. BFS is fundamental in graph processing, and many modern implementations focus on improving its performance. For example, Luo et al. [145] achieve faster execution times by leveraging GPUs.

Depth-first search. DFS [89] also starts from a chosen *root* node and visits all other nodes forming a tree-like structure, however it does so by exploring each path as deeply as possible. The algorithm uses recursion; it starts with the chosen node, visits, and processes it, flagging it as "visited". Then, it explores each unvisited connected node, by recursively calling DFS on that node.

Shortest path. It finds the shortest path between two nodes of the graph. In unweighted graphs, the goal is to find the path with the fewest nodes and relationships; for instance, Cormen et al. [146] propose an approach to exploit BFS. In weighted graphs, the objective is instead to minimize the total cost, i.e., the sum of the weights of the edges along the path. For instance, the Dijkstra algorithm [147] solves the problem iteratively. It begins by assigning a distance of zero to the starting node and infinity to all other nodes. It then iteratively explores nodes ordered by increasing distance from the start. When visiting a node, the algorithm updates the distances of its neighbors: if the sum of the current node's distance and the edge weight to a neighbor is smaller than the previously recorded distance for that neighbor, the distance is updated. The process continues until the shortest distance to the target node is determined. The Delta-Stepping algorithm [148] introduces parallelization to improve performance, while the Bellman-Ford algorithm [149,150] handles graphs with negative weights.

All-pairs shortest path. It finds the shortest path between every pair of nodes; it can be optimized to be more efficient than computing each shortest path individually. One of the most efficient approaches, proposed by Han and Takaoka [151], uses a series of matrix manipulations on the adjacency matrix describing the graph.

Yen's shortest path. This algorithm [152] finds the top k shortest paths between two nodes. It works on both unweighted and weighted graphs and uses Dijkstra's algorithm [147] when dealing with positive weights. For $k > 1$, the algorithm iteratively finds the next shortest path by introducing a deviation point — modifying or removing an edge. This adjustment allows a different path to become the shortest. The process continues until k paths are found, after which the graph is restored by undoing all introduced deviations.

A shortest path.* This algorithm [153] looks for the shortest path between two nodes combining Dijkstra's algorithm [147] with a heuristic function. Each path is constructed by selecting nodes based on the cost function $f(n) = g(n) + h(n)$, where $g(n)$ represents the actual cost to reach the node and the term $h(n)$ is a heuristic estimate of the remaining cost to the goal. The commonly used haversine heuristic calculates the distance between two points on a sphere based on their coordinates, stored as node properties. With a well-designed heuristic, A* Shortest Path typically finds the shortest path faster than Dijkstra's algorithm.

Graph edit distance. GED is used as a measure of similarity between two graphs, being defined as the minimum number of edit operations required to transform one graph into the other. The set of transformations includes the creation, deletion, or substitution of either a node or a relationship. Under weak assumptions, the A* algorithm can be exploited to find the GED between two graphs [154,155].

Minimum weight spanning tree. The MST is a subgraph that connects a node to all its reachable nodes with the minimum possible total weight along the tree's edges. The Minimum Weight K-Spanning Tree adds the additional constraint that the tree must include at most k nodes and $k-1$ edges. Prim's algorithm [156] starts from one node and repeatedly adds the edge(s) with the smallest weight connecting a node in the tree to a node outside the tree.

Minimum direct steiner tree. This algorithm finds the Steiner Tree with the minimum weight between each source and target node. The approach described in [157] iteratively finds the shortest path to one of the target nodes, sets the weight of that path to zero, and then proceeds to the next target.

Cycle detection. This algorithm aims to find all cycles within a graph, where a cycle is defined as a path that starts and ends at the same node but never visits any node twice. One of the most efficient iterative algorithms in directed graphs is proposed in [158]. It starts by initializing each node with a list containing a single path consisting of itself. In each iteration, nodes send their path lists to all outgoing neighbors. Upon receiving paths, a node checks each one: if the node is the starting point of a path, the algorithm marks it as a cycle and removes it; otherwise, if the node appears elsewhere in the path, the cycle has already been counted, so the path is discarded. This process continues until all cycles are detected or all paths are discarded.

Maximum flow problem. It refers to the problem of finding the maximum amount of flow that can navigate through the relationships of a network. In particular, based on each relationship *capacity*, these algorithms calculate the amount of flow that can be sent from a source node to a sink node. The Ford-Fulkerson algorithm [159,160] is a popular solution for the Maximum Flow problem, which iteratively looks for paths between source and sink nodes that still have available capacity, updating the flow, until no more paths can be exploited.

4. Graph data engines

Graph Data Engines can be categorized into three main groups:

- **Graph Database Management Systems** are specialized database systems designed to store, manage, and query graph-structured data efficiently. They are designed along the classical principles that inspire relational databases, typically supporting: (1) a language for defining data types, data schemes, and indexing schemes; (2) a high-level graph query language; and (3) internal subsystems for query optimization and on-line transactions.
- **Graph Processing Frameworks** are systems designed for processing and analyzing large-scale graphs efficiently by distributing computations between multiple machines (e.g., clouds). They typically consider structurally simple graphs. Some examples include Pregel [161], Apache Spark Graph [19], and Giraph [162].
- **Specialized Graph Libraries** are software tools designed for creating, manipulating, and analyzing graphs within programming languages. They are commonly used for medium-scale data analysis, as they may not be efficient for handling very large graphs. NetworkX [18], igraph [163], and graph-tool [164] are among the most used libraries.

In this survey, we focus on the first category, graph database management systems. One key distinction is that graph databases are built to manage dynamic and complex graphs, whereas graph processing frameworks and libraries typically handle simpler static graphs. In addition, graph databases offer a range of advanced features, including transactions, persistence, data independence (physical and logical), data integrity, and consistency. Graph databases offer various solutions for data analytics. For this survey, we define 'native implementations' as either a built-in library or a process that seamlessly integrates external frameworks within the database. We do not classify - as native - solutions that require users to manually export data and import it into a separate framework. The bulk creation of a graph database may require preliminary steps:

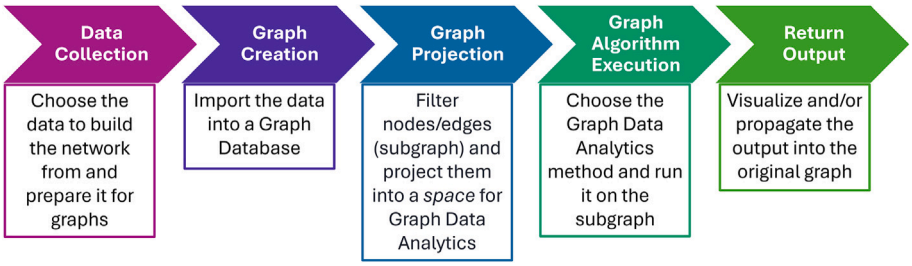

Fig. 2. Graph data analysis as an end-to-end pipeline. *1. Collect Data* and prepare it to build the network. *2. Create Graph* in the desired graph database. *3. Project Subgraph* which includes nodes and edges required for analysis, *4. Run* the graph data analytics algorithm, *5. Return output* to export the results of the analysis.

1. *Data Collection.* Select the data and design a graph schema that best suits the application. Format the data so that it can be easily uploaded into the chosen graph database.
2. *Graph Creation.* Import the network data into the graph database and create the graph.

Although graph databases usually support Online Transactional Processing (OLTP), which involves transactional queries operating on portions of the data, in this survey we concentrate on Online Analytical Processing (OLAP), i.e., on the execution of large-scale queries processing the entire graph. Fig. 2 outlines the fundamental steps to apply Analytical Processing to a graph database:

3. *Graph Projection.* Most graph databases support OLAP queries by projecting the graph into a dedicated, optimized space. This is often combined with filtering the whole graph into a subgraph to focus only on the parts needed for analysis to enhance the performance, even though this step is not strictly required.

Definition 1. *Projection.* A projection is a read-optimized, in-memory snapshot of the relevant nodes and relationships required for analysis. It serves as a computational sandbox, allowing heavy algorithmic workloads to run efficiently without impacting the latency of the main database.

4. *Run Graph Algorithm.* Choose a Graph Algorithm supported by the graph database and execute it.
5. *Return Output.* Results can be visualized directly, exported for further analysis in other systems, or integrated back into the original graph as new node or relationship properties.

In Table 3 the list of the top twenty graph database systems according to DB-Engines [10] (captured as of January 26th, 2026) is presented, reporting the supported query language and data model, how the data is stored and the name of the dedicated graph data analytics library. Among these, only those supporting the property graph data model are evaluated in greater detail; note that they almost completely overlap with the databases that support graph data analytics. We recall from the introduction that, in most systems adopting the property graph data model, machine learning tasks are mapped to generic pipelines, where the specific machine learning method is selected from a general library, without a specific effort on each method implementation. Table 4 presents the selected databases alongside the algorithms discussed in the previous section, indicating whether each database supports them and how.

4.1. Neo4j

Neo4j [165] is an open-source graph database implemented in Java, consistently recognized as the leading and most widely adopted graph database, both in research and industry settings. Neo4j is specifically designed for property graphs, storing data as a native graph database using fixed-size physical records and pointers to optimize performance; its query language, Cypher [166], serves as a foundational building block for the development of a standard Graph Query Language.

Table 3
Top 20 most used graph DBMSs according to DB-Engines [10], as of January 26th, 2026. For each system, we list the query language, the supported data models, the data storage technology, and whether they provide an internal graph data analytics (GDS) Library. In the survey, we explore the graph DBMSs supporting property graphs, highlighted in green. In the reported list, the 17th position is missing because DB-Engines assigned it to Giraph, which is a graph processing framework.

	Database	Query Language	Data Model	Storage	GDS Library
1	Neo4j	Cypher	Property Graph	Native Graph Database	Neo4j GDS
2	Azure Cosmos DB	Gremlin	Multi Model (PG included)	Document Stores	No
3	Aerospike	Gremlin	Multi Model	Tuple Stores	No
4	ArangoDB	AQL	Multi Model (PG included)	Document Stores	Graph Analytics Engines
5	Virtuoso	SPARQL	Multi Model	RDF and Relational Tables	No
6	OrientDB	SQL/Gremlin	Multi Model (PG included)	Document Stores	No
7	GraphDB	SPARQL	RDF	RDF	No
8	Amazon Neptune	Gremlin/Cypher/SPARQL	Multi Model (PG included)	RDF	Amazon Neptune Analytics
9	NebulaGraph	nGQL	Property Graph	Native Graph Database	NebulaGraph Algorithm
10	Memgraph	Cypher	Property Graph	Native Graph Database	MAGE
11	JanusGraph	Gremlin	Property Graph	Wide-Column Stores	Apache Hadoop Integration
12	Stardog	SPARQL	RDF	RDF	Stardog Spark Connector
13	TigerGraph	GSQL	Property Graph	Native Graph Database	TigerGraph GDS
14	Fauna	FQL	Multi Model	Document Stores	No
15	Surreal	SurrealQL	Multi Model	Document Stores	No
16	Dgraph	DQL	Property Graph	Key-Value Store	No
18	Blazegraph	SPARQL	Multi Model	RDF	No
19	AllegroGraph	SPARQL	Multi Model	RDF	No
20	TypeDB	TypeQL	Multi Model	Hyper-Relational Model	No

Table 4
Overview of algorithm availability on graph DBMSs that support the property graph data model. Each column represents a specific system and shows how each algorithm is supported. The considered graph DBMS versions are detailed in Appendix A. Below follows the used legend; when the technique is supported only through external implementations outside of the DBMS we use the special encoding (lowercase letters). We append '*' to indicate parallel computation and '+' to indicate incremental analytics).

Algorithm	Neo4j	Azure Cosmos DB	ArangoDB	OrientDb	Amazon Neptune	NebulaGraph	Memgraph	JanusGraph	TigerGraph	Dgraph
Degree Centrality	WB*	–	–	–	B*	B*	B	(gr)*	WB*	–
Eigenvector Centrality	WB*	–	–	–	–	–	–	(gr)*	B*	–
PageRank	WB*	–	D	–	WB*	B*	D*+	(t)*	WB*	–
HITS	B*	–	–	–	–	–	(nv)	–	–	–
Betweenness Centrality	WB*	–	WB*	–	–	WB*	B*+	(gr)*	B*	–
Closeness Centrality	B*	–	WB*	–	B*	B*	–	(gr)*	B*	–
Harmonic Centrality	B*	–	–	–	–	–	–	–	B*	–
Katz Centrality	–	–	–	–	–	–	D+	–	–	–
Influence Maximization	B*	–	–	–	–	–	–	–	WB*	–
Weakly Connected Components	WU*	–	U	–	U*	U*	U	(t)*	U*	–
Strongly Connected Components	D*	–	D	–	D*	D*	(n)	(t)*	D*	–
Label Propagation	WB*	–	D	–	WB*	B*	WB+	–	B*	–
Speaker-Listener LP	B*	–	–	–	–	–	–	–	B*	–
HANP	–	–	–	–	–	B*	–	–	–	–
Modularity Optimization	WB*	–	–	–	–	–	–	–	–	–
Louvain	WB*	–	–	–	WB*	B*	WU*	–	WU	–
K-means Clustering	✓*	–	–	–	–	–	✓	–	✓*	–
Maximum k-cut	WB*	–	–	–	–	–	–	–	–	–
Spectral Clustering	–	–	–	–	–	–	(nv)	–	–	–
K-Core Decomposition	U*	–	–	–	–	B*	(n)	–	B*	–
Biconnected Components	–	–	–	–	–	–	U	–	–	–
Graph Coloring	B*	–	–	–	–	–	WU*	–	B*	–
Maximal Independent Set	–	–	–	–	–	–	–	–	U*	–
Maximum Matching	–	–	–	–	–	–	B	–	U*	–
Triangle Counting	U*	–	–	–	–	B*	(n)	–	U*	–
K-clique	U*	–	–	–	–	–	(n)	–	–	–
Common Neighbors	B*	–	B	–	B	–	B	–	B*	–
K-Nearest Neighbors	✓*	–	–	–	–	–	–	–	✓*	–
Jaccard Similarity	WB*	–	–	–	B	B*	D	–	B*	–
Overlap Coefficient	WB*	–	–	–	B	–	D	–	–	–
Cosine Similarity	WB*	–	–	–	–	–	D	–	WB*	–
Adamic-Adar	B*	–	–	–	–	–	–	–	B*	–
Resource Allocation	B*	–	–	–	–	–	–	–	B*	–
Breadth First Search	B*	–	B*	–	B*	B*	B*	–	B*	–
Depth First Search	B*	–	B*	–	–	B*	B*	–	–	–
Shortest Path	WB*	–	WB*	–	WB*	B*	WB*	(gr)*	WB*	WB*
All Pairs Shortest Path	WB*	–	B*	–	–	–	WB*	–	WB*	–
Yen's Shortest Path	WB*	–	–	–	–	–	–	–	–	–
A* Shortest Path	WB*	–	–	–	–	–	WB*	–	WB*	–
Min. Weight Spanning Tree	WU*	–	–	–	–	–	(n)	–	WB*	–
Min. Directed Steiner Tree	WB*	–	–	–	–	–	–	–	–	–
Cycle Detection	–	–	–	–	–	–	U	(gr)*	B*	–
Maximum Flow Problem	WD*	–	–	–	–	–	WD	–	WD*	–

	––	Not Supported		✓	Supported with no special condition	(n) NetworkX
	U	Undirected Edges		WU	Weighted Undirected Edges	(nv) NVIDIA cuGraph
	D	Directed Edges		WD	Weighted Directed Edges	(gr) Apache Gremlin
	B	Both Directions		WB	Weighted Both Directions	(t) Apache Tinkerpop

Neo4j developed its own library, Neo4j Graph Data Science (GDS), to easily access and apply graph data analytics algorithms. Before using GDS, Neo4j requires the data to be loaded into an in-memory structure termed *GDS graph*. This loading step, known as *projection*, can be done in three ways:

- From a Neo4j database instance, using Cypher to load it from disk storage.
- From external sources, via an Apache Arrow [167] connection.
- From an existing GDS graph, by filtering or sampling it.

The GDS graph can be used for any Neo4j GDS procedure and can be updated or modified to support additional algorithms/tasks or to store analysis results. These updates can then be exported as external data or written as properties in the original Neo4j graph, stored in persistent memory. Neo4j GDS procedures include a specific mode designed to write results directly back to the database.

4.2. Azure cosmos DB

Azure Cosmos DB [168] is a closed-source NoSQL database developed by Microsoft that supports multiple data models, including the property graph model. Essentially, Cosmos DB is a document store, where nodes and relationships are stored as JSON documents. It enables graph traversal through the use of the Apache Gremlin [169] query language.

Azure Cosmos DB does not natively support OLAP queries and, as a result, does not inherently include graph data analytics capabilities. To perform graph data analytics algorithms on data contained in Cosmos DB, the data must first be prepared for processing. As a result, Azure Cosmos DB *projects* graph data into an external framework, which is outside the scope of this survey. However, it is worth noting that Cosmos DB can be easily connected to Azure Databricks, which supports GraphFrames [170], an Apache Spark [171] package dedicated to graphs that supports many graph data analytics algorithms.

4.3. ArangoDB

ArangoDB [172] is a closed-source, multi-model database designed primarily for RDF graphs, but it also supports property graphs. It is implemented in C++ for high performance and uses a document-based storage model. ArangoDB features its own graph query language, AQL [173], which is a declarative language closely integrated with its JSON-like storage format.

ArangoDB offers Graph Analytics Engines as a service for graph data analytics. It follows the approach of projecting data from the database into a dedicated in-memory space optimized for OLAP tasks. To use it, an engine of the required size must be created, where the entire graph from the ArangoDB database is loaded. Once the graph is projected, the Graph Analytics Engine provides various graph algorithms for data processing. After the computation, the results can be streamed back into the original graph. ArangoDB offers a function specifically to choose which calculated values to write back to specified properties.

4.4. OrientDB

OrientDB [174] is an open-source NoSQL database that supports multiple data models, including property graphs. It is implemented in Java and uses a document-based storage system. The database features an SQL-like query language and also supports Gremlin for more advanced graph traversal queries. However, OrientDB does not provide built-in solutions for graph data analytics.

4.5. Amazon Neptune

Amazon Neptune [175] is a closed-source graph database developed by Amazon Web Services (AWS) that supports both RDF graphs and property graphs. It is compatible with three popular graph query languages: Gremlin, Cypher, and SPARQL, the standard query language for RDF databases.

For graph data analytics, AWS offers Neptune Analytics as a complement to the Neptune database. Neptune Analytics is a memory-optimized graph database engine designed for analytics. It stores large graph datasets in memory, enabling low-latency queries and providing a library of optimized graph analytics algorithms. To create the in-memory graph, Neptune Analytics offers two solutions:

- From an Amazon Neptune database instance, or a snapshot of it (with a direct connection between services).
- From an Amazon S3 file, a data storage service that can be used to transfer data.

As a Neptune Analytics graph, the data can be processed using the built-in algorithms from its library. Amazon Neptune Database seamlessly integrates both the loading of the data and the reverse, allowing results to be directly streamed back into the original database with dedicated procedures.

4.6. NebulaGraph

NebulaGraph [176] is an open-source graph database, built in C++ and designed for property graphs. It features its own query language, nGQL [177], which is compatible with Cypher.

To support graph data analytics algorithms, NebulaGraph offers NebulaGraph Algorithm, a Spark application built on GraphX [19,178]. It retrieves graph data from the database using the NebulaGraph Spark Connector, and then projects it into a GraphX graph. From there, it applies graph data analytics algorithms, either from GraphX or custom implementations by NebulaGraph. While the database itself is open-source, in addition to NebulaGraph Algorithm it also provides additional algorithms for an enterprise edition whose list of covered algorithms is not publicly accessible.

NebulaGraph does not support writing results back to the graph; outputs are limited to viewing or CSV export.

4.7. Memgraph

Memgraph [179] is an open-source, high-performance property graph database built in C and C++ using Cypher as the query language. Memgraph features a hybrid architecture that allows data storage in different modes: in-memory transactional mode (for safer transactions), in-memory analytical mode (not ACID-compliant, but it enhances performance), and on-disk (for handling larger datasets).

The Memgraph team developed the Memgraph Advanced Graph Extensions (MAGE) library, an open-source repository that provides a collection of functions for graph and path querying, along with various graph data analytics algorithms and machine learning methods. Although not strictly required, Memgraph's in-memory analytical mode is often used for MAGE procedures due to the high computational demands of some algorithms. This mode is optimized for these procedures, and therefore they can be directly run on the data. Additionally, Memgraph allows users to create graph projections, which are smaller subgraphs that focus on specific parts of the graph, further improving performance. MAGE, as an open-source repository, is designed for easy integration with external sources. It supports both external libraries, such as NetworkX, and the ability to implement custom procedures in C, C++, and Python.

MAGE does not inherently support writing back results, but they can be yielded within the database environment and stored via a query.

4.8. JanusGraph

JanusGraph [176] is an open-source graph database written in Java, designed for property graphs. It natively supports Gremlin as its query language and uses wide-column stores for data storage.

For graph data analytics and, more in general, OLAP queries, JanusGraph can be integrated with Apache Hadoop [180] and Apache Spark. In particular, when setting up a JanusGraph instance, its configuration can be adjusted to connect directly to a Hadoop cluster. This enables the use, within the Gremlin console, of those graph data analytics algorithms available in the Hadoop framework. It is also natively integrated with the Apache TinkerPop [181] graph stack, leveraging its suite of graph algorithms for graph data analytics capabilities.

JanusGraph can yield results processed by Apache TinkerPop and store them in the original graph.

4.9. TigerGraph

TigerGraph [182] is a native graph database built in C++ to develop and analyze property graphs. TigerGraph is not open-source, but it offers a limited free-tier version. Its query language, GSQL [183], combines familiar SQL-like features with powerful graph traversal capabilities. Unlike other graph databases, TigerGraph has a unique constraint on the data schema: only one relationship of the same type can exist between any two nodes.

TigerGraph has developed its own graph data analytics library, TigerGraph GDS, where algorithms are written as GSQL queries and can be run directly on the data. Although performance cannot be optimized by restricting the analysis to a specific subgraph, many individual algorithms can filter nodes and relationships based on labels and types.

TigerGraph GDS algorithms support an optional parameter to specify whether and to which property the results should be written.

4.10. Dgraph

Dgraph [184] is a closed-source database designed natively for property graphs, built on a key-value storage system. It uses DQL [185], a query language that retrieves and navigates graph data in a JSON-like format. While DQL includes an implementation of the Shortest Path algorithm, Dgraph does not provide built-in support for broader graph data analytics tasks.

4.11. Other property graph database systems

While this survey prioritizes the top twenty Graph DBMSs listed by DB-Engines, other systems supporting the property graph model offer solutions for graph data analytics. These systems align with the categories previously identified: those developing native internal libraries and those relying on integrations with external analytical frameworks.

Among the systems providing extensive native support, Ultipa [186] offers a comprehensive library of graph analytics algorithms, comparable in scope to those found in Neo4j, Memgraph, and TigerGraph. To achieve high efficiency, Ultipa leverages in-memory or distributed projections of the graph data. Similarly, Oracle [187] provides graph analytics capabilities through Graph Studio within the Oracle Autonomous Database. Although the data is inherently stored in relational tables, Graph Studio allows users to remodel this data into property graphs. These graphs are then projected into a specialized in-memory graph server, granting access to an extended library of native algorithms similar to the leading property graph databases.

A different approach is taken by embedded graph databases, which run within the application process. Kùzu [188] is an embedded system that natively supports a focused set of five algorithms (PageRank, Louvain, K-Core Decomposition, Weakly Connected Components, and Strongly Connected Components), relying on projections to ensure scalability and performance. Sparksee [189] operates similarly; it is integrated at the code level across various programming languages and natively offers seven algorithms (PageRank, Weakly Connected Components, Strongly Connected Components, Scalable Community Detection, Breadth First Search, Depth First Search, Shortest Path). VelocityDB [190], an Object/NoSQL database supporting property graphs embedded in C#, does not provide a native analytics library. However, the architecture of these embedded systems (Kùzu, Sparksee, and VelocityDB) facilitates seamless access to external libraries (such as NetworkX or C# analytical tools) directly within the application's environment.

Other systems prioritize integration with large-scale processing frameworks over native algorithm development. DataStax Graph [191], which is built on top of Apache Cassandra, enables graph analytics by connecting to the Apache Spark framework. HGraphDB serves as a client layer that functions as a property graph database over Apache HBase [192]; it integrates with Apache Giraph and Apache Flink Gelly to access their algorithmic libraries. AgensGraph [193], a multi-model DBMS supporting both relational and property graph models, adopts a similar strategy by integrating with Apache Hadoop to perform scalable graph data analysis.

4.12. Algorithms and query languages

Currently, graph databases treat data analytics algorithms as external functions or procedures rather than language constructs. For instance, both in Neo4j and in Memgraph, algorithm libraries are invoked using specific procedure calls (e.g., `CALL` gds.pageRank…). Similarly, TigerGraph's GSQL runs algorithms as parameterized queries, effectively acting as function calls.

An exception to this approach is found in traversal algorithms, particularly Shortest Path, which is often integrated into the traversal ideology of graph query languages. Cypher includes specific constructs (e.g., `SHORTEST`) that let the user directly specify the matching logic of graph patterns. Gremlin is traversal-centric, allowing shortest path logic to be directly expressed as a sequence of steps. GQL, the emerging standard query language, also embraces this integration by incorporating the shortest path as a path pattern prefix, thereby explicitly including it among the extraction processes from the graph. Similarly, GSQL and nGQL have specific constructs that let users integrate shortest paths within their pattern queries.

Memgraph further extends this integration by expanding the language to other traversal algorithms. Directly within its Cypher implementation, it allows users to specify the traversal strategy (using Breadth First Search or Depth First Search) as part of the pattern construction syntax.

Looking forward, integrating more of the analytical algorithms into query languages could enhance the expressiveness of graph databases. For example, constructs that let users rank according to centrality measures (e.g., `ORDER BY PageRank`) or aggregate over communities (e.g., *GROUP BY Louvain*) could allow users to incorporate complex analytical metrics dynamically within the `MATCH` or `RETURN` clauses. In [194], de Graaf introduced GraphAlg, a recent approach for the integration of data algorithms within graph database query languages.

5. Performance evaluation

One of the main contributions of our survey is the performance comparison between different Graph DBMSs supporting the property graph model. In particular, we selected three systems – Neo4j, Memgraph, and TigerGraph – as they provide comprehensive support for graph data analytics and also feature native implementations of many algorithms, minimizing reliance on external frameworks. Additionally, they offer open-source versions for research purposes.

We ran the experiments for Neo4j and Memgraph on a server configured with a 14-core Intel Xeon E5-2660 v4 CPU, 128 GB of RAM, and SSD storage. The experiments for TigerGraph were executed on an Amazon AWS *TG2* computing instance provisioned by TigerGraph Cloud, which was equipped with 8 vCPUs, 128 GB of RAM, and SSD storage. All three databases, Neo4j, Memgraph, and TigerGraph were tested using default configurations with minimal changes to utilize all the available memory.

We chose this configuration due to TigerGraph's licensing model, which restricts full access to the graph data analytics library in the free on-premise version. Consequently, the cloud instance represents the highest-performance environment available to researchers without a paid enterprise license. While we acknowledge the architectural differences between the on-premise and cloud environments, we ensured that memory capacity was identical (128 GB) across all systems. Therefore, this comparison reflects the performance of the most capable accessible configurations for each DBMS available to the research community.

5.1. Performance evaluation on synthetic datasets

We tested Graph DBMSs on synthetic datasets [195] of increasing size while maintaining similar network characteristics, thus providing an initial indicator of the system's scalability. The datasets were generated using three different network topologies: Random, Scale-Free, and Small-World. For each topology, multiple datasets were created with increasing sizes while keeping edge density constant across different topologies and sizes. The number of nodes doubled at each step, ranging from 100,000 to 3.2 million, and the largest graphs generated contain more than 100 million edges.

For each Graph DBMS – Neo4j, Memgraph, and TigerGraph – we tested three algorithms on every synthetic dataset: PageRank, Label Propagation, and Weakly Connected Components. Each test was repeated 10 times, and the averaged results are shown in Fig. 3.

We chose these algorithms for our preliminary analysis because of their availability across many different systems. Moreover, we believe these three algorithms well represent different graph analytics tasks and different computational complexities.

Across all Graph DBMSs, algorithms run faster on Small-World networks compared to other topologies. This is likely due to their high clustering coefficients and short average path lengths, which lead to a small number of iterations needed to propagate information and short execution times.

Generally, algorithms exhibit similar execution times on Random and Scale-Free networks. However, Label Propagation performs slightly better on Random networks across all Graph DBMSs. This is likely because edges are more evenly distributed, similar to Small-World networks, allowing for fewer iterations in information propagation. PageRank, instead, runs faster on Scale-Free networks compared to Random networks. This may be because most edges are connected to a

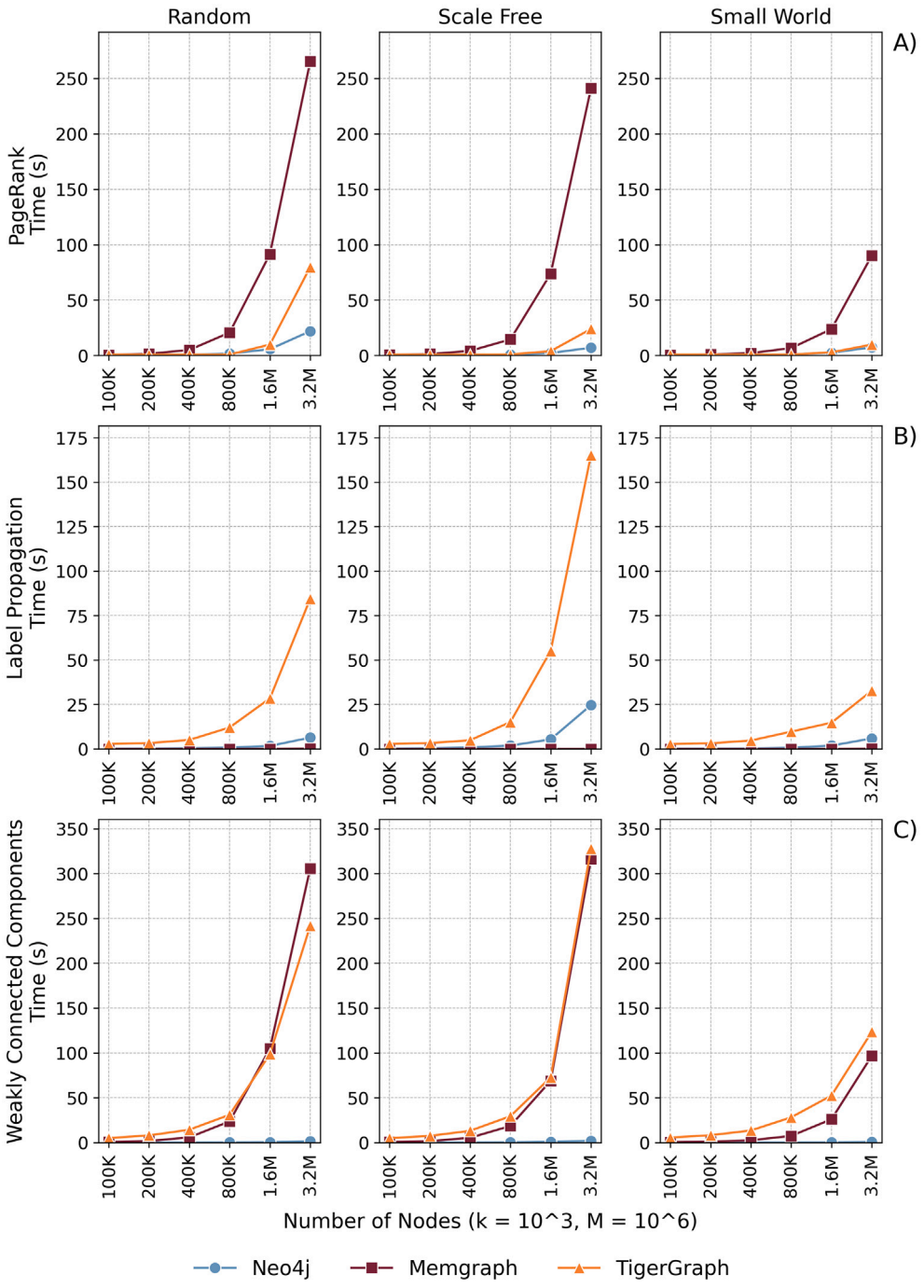

Fig. 3. Performance evaluation over synthetic graphs of different sizes and topologies (each column of the plot matrix corresponds to a specific topology: Random, Scale Free, and Small World). We consider three graph DBMSs (Neo4j, Memgraph, TigerGraph) over three algorithms: (A) PageRank, (B) Label Propagation, and (C) Weakly Connected Components. In general, Neo4j performance is superior in PageRank and Weakly Connected Components, while Memgraph is superior in Label Propagation. In the comparison between Memgraph and TigerGraph, we note that TigerGraph outperforms Memgraph in PageRank, Memgraph outperforms TigerGraph in Label Propagation, and the two systems perform very similarly in Weakly Connected Components.

few high-degree nodes, enabling the algorithm to quickly identify those important nodes and consequently compute centrality for the entire network in fewer steps.

5.2. Performance evaluation on real graphs

To further assess the performance of Graph DBMSs, we tested seven widely used graph data analytics algorithms on five real-world graph datasets, each with unique network characteristics.

5.2.1. Benchmark datasets

The structure of graph data is one of the most significant factors affecting performance. Therefore, a comprehensive evaluation must consider graphs with diverse characteristics. In [196], a collection of graph datasets built from real-world data is proposed, as a benchmark for machine learning; we selected five of these datasets, based on their network characteristics reported in Table 5.

Table 5
General network features of the benchmark graph dataset used for performance evaluation.

Graph Dataset	Nodes	Edges	Density	Average Node Degree	Average Clustering Coefficient	Graph Diameter	Special Features
ArXiv	169,343	1,166,243	4×10^{-5}	13.7	0.226	23	Random network topology
Bio-KG	93,773	5,088,434	6×10^{-4}	47.5	0.409	8	Scale-Free network topology
DDI	4267	1,334,889	0.14	500.5	0.514	5	Small-World network topology
Collab	235,868	1,285,465	4×10^{-5}	8.2	0.729	22	Strongly connected communities
HIV	1,049,162	1,129,688	2×10^{-6}	2.2	0.002	12	Many disconnected components

ArXiv. This dataset is built from the ArXiv [197] citation network of Computer Science articles. It forms a homogeneous directed graph, where each node represents an article and is linked to the articles it cites. The resulting graph consists of 169,343 nodes and 1,166,243 edges, with a density of $4x10^{-5}$, making it highly sparse.

The graph has an average node degree of 13.7. While the degree distribution appears to follow a power law, it decays very rapidly, as only a few nodes have significantly more connections than others. As a result, the graph does not exhibit a truly scale-free structure. The graph has a clustering coefficient of 0.226 (indicating that papers rarely form tightly connected communities), and a diameter of 23, which is of the same order of magnitude as the logarithm of the number of nodes ~ 5. According to the Erdös–Rényi model [57], this suggests that the graph's structure is similar to that of a random network.

Bio knowledge graph. The Bio Knowledge Graph (Bio-KG) dataset is created by merging multiple biomedical data repositories, resulting in a heterogeneous, directed graph. It consists of five types of nodes: Disease (10,687 nodes), Proteins (17,499), Drugs (10,533), Side Effects (9,969), and Protein Function (45,085). The graph includes 51 different types of relationships that capture various biological connections. For example, it features 38 types of Drug-Drug interactions, 8 types of Protein-Protein interactions, as well as relationships between Drugs and Proteins, Drugs and Side Effects, and Protein Functions.

Bio-KG contains a total of 93,773 nodes and 5,088,434 edges, resulting in a density of $6x10^{-4}$. This indicates a highly sparse graph. However, it is worth noting that its density is significantly higher compared to other datasets. The graph has an average clustering coefficient of 0.409 and a diameter of 8, showing overall a good level of connectivity among nodes; it has an average node degree of 47.5 and its node degree distribution follows the power law, resulting in a Scale-Free structure of the network.

Drug-drug interaction. The Drug-Drug Interaction (DDI) graph is undirected and homogeneous, based on the database presented in [198], which captures interactions between drugs. In this graph, nodes represent either FDA-approved or experimental drugs, while each edge indicates a drug interaction, i.e., when taking two drugs together produces a significantly different effect compared to taking each drug individually. The resulting graph has 4267 nodes and 1,334,889 relationships, with a density of 0.14.

While this density is generally considered low, DDI is significantly denser compared to other datasets, primarily due to its smaller number of nodes. The average node degree is 500.5, and the overall degree distribution shows that edges are more evenly distributed across nodes, partly due to the higher density. The graph also has a relatively high average clustering coefficient of 0.514 and a diameter of 5. These characteristics suggest that nodes tend to form tight-knit groups, and any node can reach others in just a few steps. Based on these parameters, the network structure resembles that of a Small-World network.

Collab. This dataset is an undirected, homogeneous graph constructed from a subset of the Microsoft Academic Graph (MAG). Nodes represent authors, and edges capture their collaboration network. The resulting graph consists of 235,868 nodes and 1,285,465 edges, with a density of $4x10^{-5}$, indicating a highly sparse network. The graph has an average node degree of 8.2, and its degree distribution follows a lognormal decay. This means that edges are more evenly distributed across the network.

While some nodes have more connections than others, the difference is not extreme, and there are no overwhelmingly dominant hubs. The average clustering coefficient is 0.729, which is unusually high for a real-world network. This suggests that authors tend to collaborate in tightly-knit groups, where an author's co-authors are also likely to collaborate with each other. In contrast, the graph diameter is 22, which is relatively large. This indicates a sparse network with limited shortcuts between distant authors, suggesting that no highly connected nodes act as bridges to efficiently link different parts of the network. Therefore, the network structure corresponds to a strongly connected graph with several subgraphs representing loosely connected communities.

Molecule HIV. This graph is derived from one of the MoleculeNet [199] datasets, resulting in a disconnected, undirected, and homogeneous graph. Each connected component represents a molecule, where nodes correspond to atoms and edges represent chemical bonds. In total, the graph consists of 41,127 components, 1,049,162 nodes, and 1,129,688 edges, with a density of $2x10^{-6}$, indicating an extremely sparse network. This is due to the high number of components, each containing fewer than 26 nodes on average. This is also reflected in the low average node degree of 2.2, a flat degree distribution overall, and a very low average clustering coefficient of 0.002. The graph has a diameter of 12, but due to the large number of disconnected components, this value is not a meaningful representation of the overall network structure.

5.2.2. Performance evaluation

Figs. 4 and 5 present the results of our performance evaluation. Based on their popularity and their availability across different Graph DBMSs, we selected seven graph data analytics algorithms: (1) PageRank, (2) Betweenness Centrality, (3) Label Propagation, (4) Weakly Connected Components, (5) Jaccard Similarity, (6) Shortest Path, and (7) Breadth First.

We selected popular algorithms from various categories to ensure that they would be shared across the systems. The exception is Jaccard Similarity: while all three systems support it, TigerGraph does not allow for single-pair calculations, making the test not comparable with other DBMSs. This limitation applies to other similarity algorithms in TigerGraph as well. However, we included Jaccard Similarity to represent the Similarity category of algorithms.

These algorithms were tested on the five benchmark datasets presented. To ensure statistical significance, each test was repeated 10 times, and the execution times were averaged. Additionally, since the last three algorithms (Jaccard Similarity, Shortest Path, and Breadth First) depend on specific node selections, the nodes were randomly chosen in advance and kept consistent across all Graph DBMS evaluations. Specifically, for Jaccard Similarity and Shortest Path, we tested 100 pairs of nodes per run, while for Breadth First, we selected 25 random starting nodes.

5.2.3. Discussion

General remarks. The execution times of the tested algorithms are generally close, except for Betweenness Centrality, which is the heaviest (it may require hours); Jaccard Similarity is the fastest, consistently

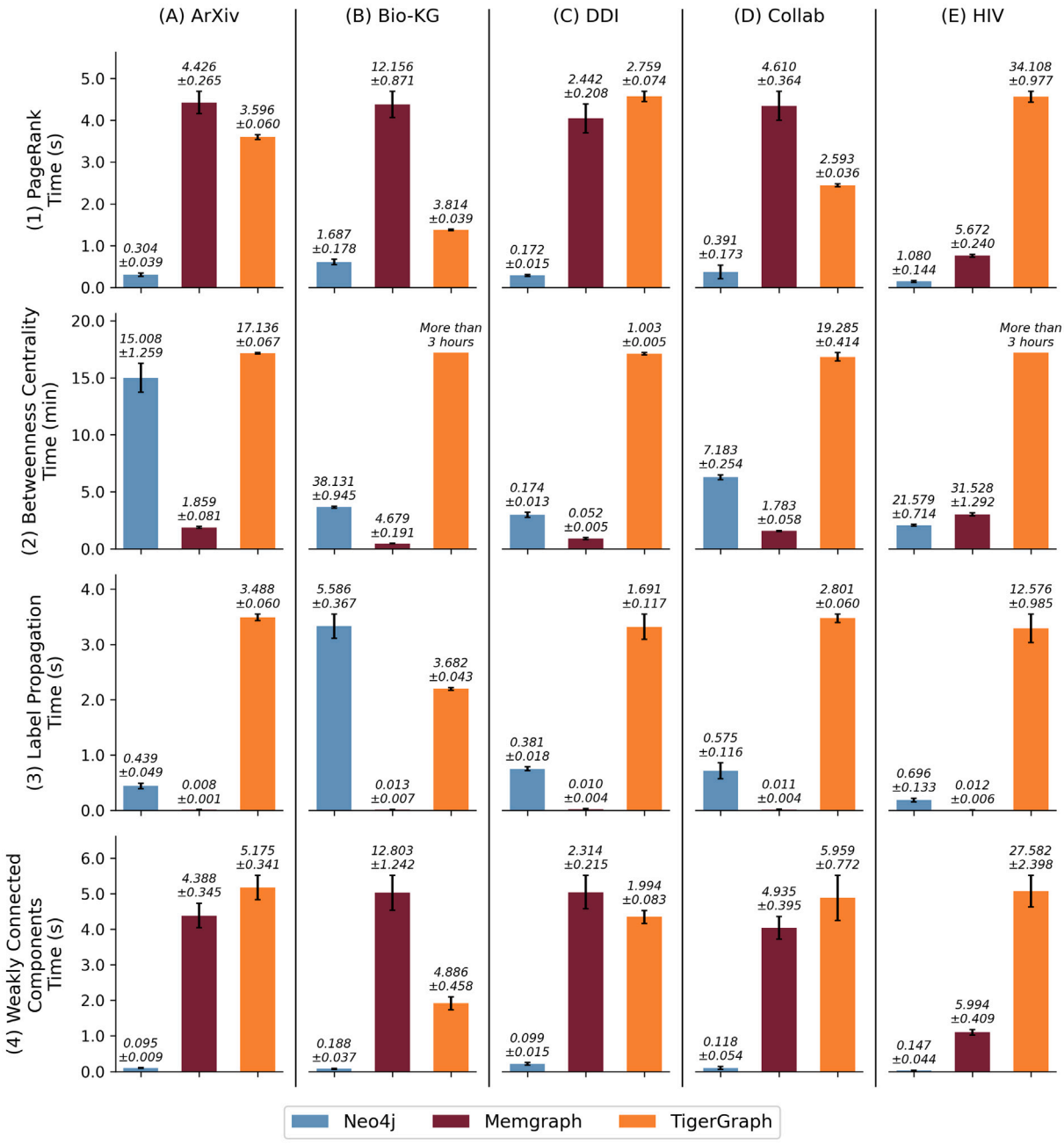

Fig. 4. Performance evaluation of various graph data analytics algorithm implementations across three graph DBMSs. Each row represents a different algorithm: (1) PageRank, (2) Betweenness Centrality, (3) Label Propagation, (4) Weakly Connected Components. The columns correspond to five benchmark datasets: (A) ArXiv, (B) Bio-KG, (C) DDI, (D) Collab, and (E) HIV. For each test, the average value, the standard deviation, and error bars of the times measured are reported.

requiring less than one second. By comparing total execution times on the five datasets, BioKG and HIV require the longest execution times, likely due to their larger number of nodes and edges, while the DDI dataset requires the shortest execution time, likely due to its significantly smaller number of nodes.

The low standard deviation and error bars indicate that the measurements are consistent. Therefore, we can reliably use the average times for further consideration.

When comparing different database systems, TigerGraph generally exhibits the worst performance in many use cases. Meanwhile, Neo4j and Memgraph outperform each other depending on the dataset and algorithm, showing variable performance across different scenarios. Notably, TigerGraph performs particularly poorly on HIV, suggesting it is more affected by a high number of nodes, while Neo4j in some cases has worse performance with BioKG, indicating that its performance is worse with a high number of edges.

PageRank. The first row of Fig. 4 presents the execution times of PageRank; Neo4j consistently outperforms the others across all five benchmark datasets, while Memgraph and TigerGraph exhibit similar execution times overall (in accordance with the experiment on synthetic datasets).

Betweenness centrality. In four out of five benchmark datasets, Memgraph significantly outperforms both Neo4j and TigerGraph. In the

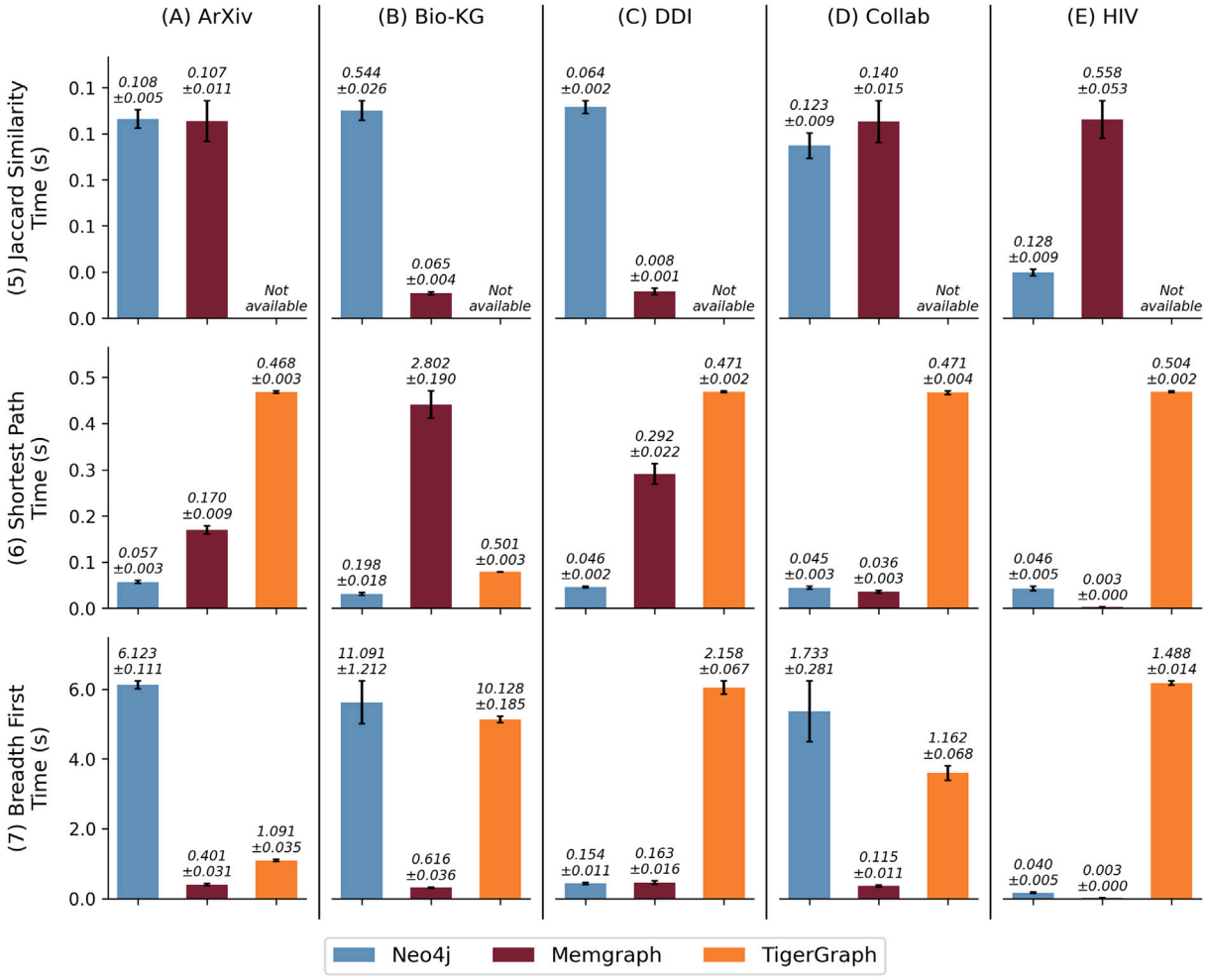

Fig. 5. Performance evaluation of various graph data analytics algorithm implementations across three graph DBMSs. Each row represents a different algorithm: (5) Jaccard Similarity, (6) Shortest Path, and (7) Breadth First. The columns correspond to five benchmark datasets: (A) ArXiv, (B) Bio-KG, (C) DDI, (D) Collab, and (E) HIV. For each test, the average value, the standard deviation, and error bars of the times measured are reported.

HIV benchmark, both Memgraph and TigerGraph exhibit slower execution times, with Memgraph taking more than 10 times longer w.r.t. the other benchmark datasets. This can be explained by the fact that Betweenness Centrality requires computing the shortest paths between all node pairs, making its execution time highly sensitive to the number of nodes in the graph. Neo4j exhibits execution times comparable to those in other benchmark datasets, suggesting that its implementation may benefit from the high number of disconnected components in the HIV graph.

Label propagation. Memgraph consistently achieves the fastest performance across all benchmark datasets, completing in a fraction of a second and appearing unaffected by variations in network structure. Neo4j outperforms TigerGraph in most tests, except for the Bio-KG dataset.

Weakly connected components. Neo4j consistently outperforms both Memgraph and TigerGraph, maintaining similar execution times across all datasets. In contrast, Memgraph and TigerGraph exhibit comparable performance, with the most significant differences appearing, as expected, in the Bio-KG and HIV datasets.

Jaccard similarity. In the first row of Fig. 5, the execution times of the Jaccard Similarity Algorithm are presented. While TigerGraph supports Jaccard Similarity, its implementation does not allow for pairwise comparisons of selected nodes. Instead, it either compares a node to all others in the graph or computes all possible pairwise comparisons. This approach is not compatible with the tests performed on Neo4j and Memgraph, where the algorithm is executed on a specific pair of nodes. Therefore, the data for TigerGraph is not available. As previously observed, Memgraph's execution times appear to be highly sensitive to the number of nodes in the graph, as they increase in line with the node count across different datasets. In contrast, Neo4j performance is more sensitive to the number of edges within the graph.

Shortest path. The second row presents the execution times for the Shortest Path algorithm, revealing distinct performance patterns across the three graph database systems. Neo4j maintains consistently low execution times, with only a slight increase, about four times slower, on the Bio-KG dataset. Memgraph, however, exhibits the highest variability, suggesting its performance is strongly influenced by different real-world network topologies. For example, in the HIV dataset, it significantly outperforms the other databases, likely benefiting from the high number of disconnected components. On the other hand, in the Bio-KG dataset, Memgraph performs notably worse than both Neo4j and TigerGraph, possibly due to the higher number of edges. Finally, TigerGraph appears unaffected by network topology, displaying consistent execution times with minimal variation across all datasets. Interestingly, graph diameter, which measures the longest shortest path between two nodes, does

not seem to correlate with execution times. A smaller diameter would suggest that fewer steps are needed to find the shortest path; yet, the DDI dataset, which has the lowest diameter among all datasets, does not show better performance than the others.

Breadth first. Memgraph is the fastest across all datasets, and, similar to the Shortest Path test, its execution time remains remarkably low. This suggests that Memgraph optimizes traversal algorithms to efficiently handle disconnected components, further reinforcing its advantage in such scenarios. Neo4j exhibits significant variability across different datasets. While it achieves low execution times for DDI and HIV, its performance declines considerably for ArXiv, Bio-KG, and Collab. The slowdown in Bio-KG can be attributed to the higher number of edges, similar to other algorithms. However, for ArXiv and Collab, the likely cause is their large graph diameter, which negatively impacts Neo4j's performance by requiring the extraction of longer paths. TigerGraph demonstrates consistent execution times across most datasets, mirroring its behavior in the Shortest Path test, except for a noticeable slowdown in Bio-KG.

6. Conclusion

Graphs are quite powerful in representing various domains of research, and graph databases have emerged as an effective way to model and query complex network data.

6.1. Contributions

In this survey, we describe how data analytics algorithms are supported by property graph databases, as they provide efficient solutions for effectively managing and analyzing graph data. We first examine a comprehensive list of algorithms frequently offered by different graph databases, which are clustered into seven categories (centrality, community, cohesiveness, covering, pattern matching, similarity, and traversal). Next, we explore how property graph databases implement these algorithms efficiently, along with a summary of which systems support each algorithm.

Most property graph database management systems support graph data algorithms, but their solutions take significantly different directions. The first approach is to develop a built-in library, making this strategy very accessible and easy to use; among them, Neo4j and TigerGraph offer comprehensive solutions, while Amazon Neptune Analytics and ArangoDB's Graph Analytical Engines offer more limited solutions. The second approach is integrating data management systems with external frameworks for data analytics; among them, Azure Cosmos DB allows data export to the Azure environment and JanusGraph to Apache Hadoop. The third approach is hybrid, as it combines both the development of internal libraries and access to external sources. Memgraph's MAGE includes a large set of built-in algorithms in addition to access to external libraries, while NebulaGraph offers a smaller set of native algorithms but provides integration with Spark's GraphX.

Finally, we focus on three property graph databases - Neo4j, Memgraph, and TigerGraph - offering accessible and comprehensive solutions; we evaluate their performance in implementing popular algorithms on both synthetic datasets — spanning various topologies and sizes — and real-world network datasets. In terms of performance, Neo4j and Memgraph appear to be the most performing systems, but execution times vary significantly depending on the graph size, the graph features, and the algorithm tested.

6.2. Limitations of the study

Our work aims to provide an overview of the landscape of graph data analytics within graph database management systems; however, it is subject to specific boundaries and limitations.

The scope of this survey is strictly focused on graph data analytics algorithms, explicitly excluding machine learning pipelines and graph learning techniques. This decision is motivated by the current state of technology: while graph databases provide well-founded strategies for analytic algorithms, machine learning and graph learning tasks are typically not consolidated within database systems. Therefore, including them would have expanded the scope beyond the focus on database-native capabilities.

Then, we focused our analysis on the top-ranked systems according to usage rankings to ensure our findings remain relevant to the largest possible community of active users. We also restricted our deeper analysis to graph databases that support the property graph data model. As highlighted in our mapping of systems to algorithms, there is a significant overlap between systems adopting the property graph model and those providing native support for graph data analytics.

Finally, our performance evaluation was strictly limited to systems offering a free or open-source version (Neo4j, Memgraph, and TigerGraph). We excluded enterprise-only solutions for licensing reasons and to provide a practical roadmap for researchers, developers, and practitioners who may not have access to commercial licenses, ensuring that our results are reproducible and broadly applicable.

6.3. Outlook

Users choosing a graph database should carefully consider their application, e.g., the type of analysis they need to perform. Our analysis guides the choice of the best solution for different applications and settings.

We lay the foundations for the guidelines for a design space for analytics in graph databases. A critical step in this workflow is the efficient transfer of data from transactional storage to an environment optimized for analytics. This concept of *projection*, and its interaction with the persistent graph, serves as the pivot upon which effective graph analytics relies.

By organizing these algorithms into a clear taxonomy and detailing how current systems support them, we have identified critical gaps and emerging opportunities for the future of graph data analytics.

As shown in Table 4, support for algorithms is highly variable. While centrality and community detection are widely supported, other categories are more neglected. Offering a more balanced assortment of algorithms across diverse categories would enable users to conduct more detailed analyses, significantly enhancing the utility of these systems.

Another significant opportunity exists to integrate analytics directly into languages like Cypher, GQL, or GSQL. Currently, most algorithms are treated as external functions rather than first-class citizens of the query language. Taking inspiration from how some query languages integrate specific traversal algorithms, future standards could allow users to use algorithmic results dynamically within queries.

Finally, regarding practical future steps, the community needs to study additional systems in greater detail. There is a need to understand how these other systems handle graph analytics and to perform technical performance evaluations, including on the closed-source solutions listed but not yet tested. Beyond the scope of this survey, there is a need to expand efforts into other areas of graph data science. For instance, while this survey focused on the central analytics workflow, future research should investigate how different databases handle data ingestion and result writing. Another promising direction involves understanding how graph databases support machine learning and graph learning tasks.

CRediT authorship contribution statement

Francesco Cambria: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft. **Francesco Invernici:** Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Writing – review and editing. **Anna Bernasconi:** Conceptualization, Resources, Supervision, Writing – review and editing. **Stefano Ceri:** Conceptualization, Resources, Supervision, Writing – review and editing.

Declaration of competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This paper is supported by the FAIR (Future Artificial Intelligence Research) project, funded by the NextGenerationEU program within the PNRR-PE-AI scheme (M4C2, Investment 1.3, Line on Artificial Intelligence).

Appendix A. System specification

Versions of the considered graph databases and the related GDS libraries are reported in Table A.6.

Table A.6
Versions of the graph database systems and GDS libraries studied in this survey. Since Azure Cosmos DB does not use release numbers, we evaluated the version available as of January 26th, 2026.

Graph Database	DBMS Version	GDS Library Version
Neo4j	5.26	2.25
Azure Cosmos DB	–	–
ArangoDB	3.12	3.12
OrientDB	3.2.48	–
NebulaGraph	3.8	3.8
Amazon Neptune	1.4.6.3	1.4.6.3
Memgraph	3.7.2	3.7.2
JanusGraph	1.1	1.1
TigerGraph	4.2	3.10
Dgraph	25.1	–

Data availability

Data will be made available on request.

References

[1] G. Fletcher, J. Hidders, J.L. Larriba-Pey, et al., Graph Data Management, Springer, 2018.
[2] A.-L. Barabási, Network science, Philos. Trans. R. Soc. A: Mathematical, Physical and Engineering Sciences 371 (1987) (2013) 20120375.
[3] P.E.N. Lutu, Using Twitter mentions and a graph database to analyse social network centrality, in: 2019 6th International Conference on Soft Computing & Machine Intelligence (ISCMI), IEEE, 2019, pp. 155–159.
[4] S. Timón-Reina, M. Rincón, R. Martínez-Tomás, An overview of graph databases and their applications in the biomedical domain, Database 2021, baab026.
[5] I. Mazein, A. Rougny, A. Mazein, R. Henkel, L. Gütebier, L. Michaelis, M. Ostaszewski, R. Schneider, V. Satagopam, L.J. Jensen, et al., Graph databases in systems biology: a systematic review, Brief. Bioinform. 25 (6) (2024) bbae561.
[6] F. Invernici, A. Bernasconi, S. Ceri, Searching Covid-19 clinical research using graph queries: algorithm development and validation, J. Med. Internet Res. 26 (2024) e52655.
[7] L. Bellomarini, L. Bencivelli, C. Biancotti, L. Blasi, F.P. Conteduca, A. Gentili, R. Laurendi, D. Magnanimi, M.S. Zangrandi, F. Tonelli, et al., Reasoning on company takeovers: from tactic to strategy, Data Knowl. Eng. 141 (2022) 102073.
[8] A. Colombo, A. Bernasconi, S. Ceri, An llm-assisted etl pipeline to build a high-quality knowledge graph of the Italian legislation, Inf. Process. Manag. 62 (4) (2025) 104082.
[9] A. Colombo, F. Cambria, F. Invernici, et al., Legislative knowledge management with property graphs, in: Proceedings of the Workshops of the EDBT/ICDT 2025 Joint Conferenceco-Located with the EDBT/ICDT 2025 Joint Conference, (25 March 2025), vol. 3946, CEUR-WS.org, Barcelona, Spain, 2025, pp. 1–8.
[10] DB-Engine, 2026, Retrieved from https://db-engines.com/en/ranking/graph+dbms (26 January 2026).
[11] I. Robinson, J. Webber, E. Eifrem, Graph Databases: New Opportunities for Connected Data, O'Reilly Media, Inc., 2015.
[12] R. Angles, The property graph database model, in: AMW, 2018.
[13] P. Naur, Concise Survey of Computer Methods, Studentlitteratur, 1974.
[14] F. Provost, T. Fawcett, Data science and its relationship to big data and data-driven decision making, Big data 1 (1) (2013) 51–59.
[15] D. Donoho, 50 years of data science, J. Comput. Graph. Stat. 26 (4) (2017) 745–766.
[16] L. Cao, Data science: a comprehensive overview, ACM Comput. Surv. 50 (3) (2017) 1–42.
[17] A. Bonifati, M.T. Ozsu, Y. Tian, H. Voigt, W. Yu, E. Zhang, A roadmap to graph analytics, ACM SIGMOD Rec. 53 (4) (2025) 43–51.
[18] A. Hagberg, P.J. Swart, D.A. Schult, Exploring Network Structure, Dynamics, and Function Using Networkx, Tech. rep., Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2008.
[19] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, I. Stoica, {GraphX}: graph processing in a distributed dataflow framework, in: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), 2014, pp. 599–613.
[20] J. Leskovec, R. Sosič, Snap: a general-purpose network analysis and graph-mining library, ACM Trans. Intell. Syst. Technol. 8 (1) (2016) 1–20.
[21] U. Brandes, Network Analysis: Methodological Foundations, vol. 3418, Springer Science & Business Media, 2005.
[22] C.T. Butts, Revisiting the foundations of network analysis, Science 325 (5939) (2009) 414–416.
[23] D. Hevey, Network analysis: a brief overview and tutorial, Health psychology and behavioral medicine 6 (1) (2018) 301–328.
[24] S.P. Borgatti, A. Mehra, D.J. Brass, G. Labianca, Network analysis in the social sciences, Science 323 (5916) (2009) 892–895.
[25] J. Scott, What is Social Network Analysis? Bloomsbury Academic, 2012.
[26] G.A. Pavlopoulos, M. Secrier, C.N. Moschopoulos, T.G. Soldatos, S. Kossida, J. Aerts, R. Schneider, P.G. Bagos, Using graph theory to analyze biological networks, Biodata Min. 4 (2011) 1–27.
[27] M. Koutrouli, E. Karatzas, D. Paez-Espino, G.A. Pavlopoulos, A guide to conquer the biological network era using graph theory, Front. Bioeng. Biotechnol. 8 (2020) 34.
[28] W. Jiang, W. Ye, X. Tan, Y.-J. Bao, Network-based multi-omics integrative analysis methods in drug discovery: a systematic review, Biodata Min. 18 (1) (2025) 27.
[29] T. Pourhabibi, K.-L. Ong, B.H. Kam, Y.L. Boo, Fraud detection: a systematic literature review of graph-based anomaly detection approaches, Decis. Support Syst. 133 (2020) 113303.
[30] P. Irofti, A. Pătraşcu, A. Băltoiu, Fraud detection in networks, in: Enabling AI Applications in Data Science, Springer, 2020, pp. 517–536.
[31] B. Deprez, T. Vanderschueren, B. Baesens, T. Verdonck, W. Verbeke, Network analytics for anti-money laundering—a systematic literature review and experimental evaluation, INFORMS J. Data Sci. Vol. Articles in Advance (2025) 1–36.
[32] A. Háznagy, I. Fi, A. London, T. Nemeth, Complex network analysis of public transportation networks: a comprehensive study, in: 2015 International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS), IEEE, 2015, pp. 371–378.
[33] Q. Zhang, Z. Ma, P. Zhang, E. Jenelius, Mobility knowledge graph: review and its application in public transport, Transportation 52 (3) (2025) 1119–1145.
[34] S. Heidari, Y. Simmhan, R.N. Calheiros, R. Buyya, Scalable graph processing frameworks: a taxonomy and open challenges, ACM Computing Surveys (CSUR) 51 (3) (2018) 1–53.
[35] L. Meng, Y. Shao, L. Yuan, L. Lai, P. Cheng, X. Li, W. Yu, W. Zhang, X. Lin, J. Zhou, A survey of distributed graph algorithms on massive graphs, ACM Comput. Surv. 57 (2) (2024) 1–39.
[36] M.E. Coimbra, A.P. Francisco, L. Veiga, An analysis of the graph processing landscape, J. Big Data 8 (1) (2021) 55.
[37] Y. Jiang, A survey of task allocation and load balancing in distributed systems, IEEE Trans. Parallel Distrib. Syst. 27 (2) (2015) 585–599.
[38] R. Angles, C. Gutierrez, Survey of graph database models, ACM Computing Surveys (CSUR) 40 (1) (2008) 1–39.
[39] R. Angles, A comparison of current graph database models, in: 2012 IEEE 28th International Conference on Data Engineering Workshops, IEEE, 2012, pp. 171–177.
[40] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, D. Vrgoč, Foundations of modern query languages for graph databases, ACM Computing Surveys (CSUR) 50 (5) (2017) 1–40.
[41] H. Kondylakis, S. Dumbrava, M. Lissandrini, N. Yakovets, A. Bonifati, V. Efthymiou, G. Fletcher, D. Plexousakis, R. Tommasini, G. Troullinou, et al., Property graph standards: state of the art & open challenges, Proc. VLDB Endow. 18 (12) (2025) 5477–5481.
[42] A. Bonifati, I. Holubová, A. Prat-Pérez, S. Sakr, Graph generators: state of the art and open challenges, ACM computing surveys (CSUR) 53 (2) (2020) 1–30.
[43] A. Bonifati, G. Fletcher, H. Voigt, N. Yakovets, Querying Graphs, Morgan & Claypool Publishers, 2018.
[44] J.I. Lopez-Veyna, I. Castillo-Zuñiga, M. Ortiz-Garcia, A review of graph databases, in: International Conference on Software Process Improvement, Springer, 2022, pp. 180–195.
[45] M. Besta, R. Gerstenberger, E. Peter, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, T. Hoefler, Demystifying graph databases: analysis and taxonomy of data organization, system designs, and graph queries, ACM Comput. Surv. 56 (2) (2023) 1–40.
[46] D.B. West, et al., Introduction to Graph Theory, vol. 2, Prentice Hall Upper Saddle River, 2001.
[47] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, et al., Graph pattern matching in gql and sql/pgq, in: Proceedings of the 2022 International Conference on Management of Data, 2022, pp. 2246–2258.
[48] I.C. Secretary, Information technology - database languages - GQL. Standard ISO/IEC WD 39075, 2024. https://www.iso.org/standard/76120.html.

[49] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, A. Green, J. Hidders, B. Li, L. Libkin, V. Marsault, W. Martens, et al., Pg-schema: schemas for property graphs, Proc. ACM Manag. Data 1 (2) (2023) 1–25.

[50] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, K.W. Hare, J. Hidders, V.E. Lee, B. Li, L. Libkin, W. Martens, et al., Pg-keys: keys for property graphs, in: Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 2423–2436.

[51] S. Ceri, A. Bernasconi, A. Gagliardi, Reactive knowledge management, in: 2024 IEEE 40th International Conference on Data Engineering (ICDE), IEEE, 2024, pp. 5574–5582.

[52] S. Ceri, A. Bernasconi, A. Gagliardi, D. Martinenghi, L. Bellomarini, D. Magnanimi, PG-triggers: triggers for property graphs, in: Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 373–385.

[53] D. Magnanimi, A. Colombo, L. Bellomarini, A. Bernasconi, S. Ceri, D. Martinenghi, Enabling light-weight reasoning via cypher triggers, in: 2025 IEEE 41st International Conference on Data Engineering (ICDE), IEEE Computer Society, 2025, pp. 4277–4290.

[54] F. Cambria, F. Invernici, A. Bernasconi, S. Ceri, Mine graph rule: a new gql operator for mining association rules in property graph databases, VLDB J. 34 (4) (2025) 1–21.

[55] World Wide Web Consortium, et al. RDF 1.1 primer, World Wide Web Consortium, 2014.

[56] A. Poulovassilis, P. McBrien, A general formal framework for schema transformation, Data Knowl. Eng. 28 (1) (1998) 47–71.

[57] P. Erd6s, A. Rényi, On the evolution of random graphs, publ, Math. Inst. Hungar. Acad. Sci 5 (1960) 17–61.

[58] D.J. Watts, S.H. Strogatz, Collective dynamics of 'small-world' networks, Nature 393 (6684) (1998) 440–442.

[59] D. Dominguez-Sal, N. Martinez-Bazan, V. Muntes-Mulero, P. Baleta, J.L. Larriba-Pey, A discussion on the design of graph database benchmarks, in: Technology Conference on Performance Evaluation and Benchmarking, Springer, 2010, pp. 25–40.

[60] F.A. Rodrigues, A mathematical modeling approach from nonlinear dynamics to complex systems, in: Network Centrality: an Introduction, Springer, 2019, pp. 177–196.

[61] A. Saxena, S. Iyengar, Centrality measures in complex networks: A survey, arXiv preprint, 2020, arXiv:2011.07190.

[62] F. Bloch, M.O. Jackson, P. Tebaldi, Centrality measures in networks, Social Choice and Welfare 61 (2) (2023) 413–453.

[63] G.F. De Arruda, A.L. Barbieri, P.M. Rodriguez, F.A. Rodrigues, Y. Moreno, L.D. F. Costa, Role of centrality for the identification of influential spreaders in complex networks, Phys. Rev. E 90 (3) (2014) 032812.

[64] S. Derrible, Network centrality of metro systems, PLOS ONE 7 (7) (2012) e40575.

[65] L.C. Freeman, Centrality in social networks conceptual clarification, Soc. Netw. 1 (3) (1978) 215–239.

[66] T. Opsahl, F. Agneessens, J. Skvoretz, Node centrality in weighted networks: generalizing degree and shortest paths, Soc. Netw. 32 (3) (2010) 245–251.

[67] B. Ruhnau, Eigenvector-centrality—a node-centrality? Soc. Netw. 22 (4) (2000) 357–365.

[68] S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine, Computer networks and ISDN systems 30 (1–7) (1998) 107–117.

[69] D. Fogaras, B. Rácz, K. Csalogány, T. Sarlós, Towards scaling fully personalized pagerank: algorithms, lower bounds, and experiments, Internet Mathematics 2 (3) (2005) 333–358.

[70] B. Bahmani, A. Chowdhury, A. Goel, Fast incremental and personalized pagerank, arXiv preprint, 2010, arXiv:1006.2880.

[71] P. Lofgren, S. Banerjee, A. Goel, Personalized pagerank estimation and search: a bidirectional approach, in: Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, 2016, pp. 163–172.

[72] J.M. Kleinberg, Authoritative sources in a hyperlinked environment, Journal of the ACM (JACM) 46 (5) (1999) 604–632.

[73] M. Barthelemy, Betweenness centrality in large complex networks, Eur. Phys. J. B 38 (2) (2004) 163–168.

[74] U. Brandes, A faster algorithm for betweenness centrality, J. Math. Sociol. 25 (2) (2001) 163–177.

[75] U. Brandes, C. Pich, Centrality estimation in large networks, Int. J. Bifurc. Chaos 17 (7) (2007) 2303–2318.

[76] E. Cohen, D. Delling, T. Pajor, R.F. Werneck, Computing classic closeness centrality, at scale, in: Proceedings of the Second ACM Conference on Online Social Networks, 2014, pp. 37–50.

[77] S. Wasserman, Social Network Analysis: Methods and Applications, The Press Syndicate of the University of Cambridge, 1994.

[78] M. Marchiori, V. Latora, Harmony in the small-world, physica a: statistical mechanics and its applications, Physica A 285 (3–4) (2000) 539–546.

[79] L. Katz, A new status index derived from sociometric analysis, Psychometrika 18 (1) (1953) 39–43.

[80] A.V.D. Grinten, E. Bergamini, O. Green, D.A. Bader, H. Meyerhenke, Scalable katz ranking computation in large static and dynamic graphs, arXiv preprint, 2018, arXiv:1807.03847.

[81] D. Kempe, J. Kleinberg, É. Tardos, Maximizing the spread of influence through a social network, in: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2003, pp. 137–146.

[82] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, N. Glance, Cost-effective outbreak detection in networks, in: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2007, pp. 420–429.

[83] S. Fortunato, Community detection in graphs, Phys. Rep. 486 (3–5) (2010) 75–174.

[84] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, X. Lin, A survey of community search over big graphs, VLDB J. 29 (1) (2020) 353–392.

[85] M. Speriosu, N. Sudan, S. Upadhyay, J. Baldridge, Twitter polarity classification with label propagation over lexical links and the follower graph, in: Proceedings of the First Workshop on Unsupervised Learning in NLP, 2011, pp. 53–63.

[86] K. Ikeda, G. Hattori, C. Ono, H. Asoh, T. Higashino, Twitter user profiling based on text and community mining for market analysis, Knowl.-based Syst. 51 (2013) 35–47.

[87] R.L. Graham, D.E. Knuth, T.S. Motzkin, Complements and transitive closures, Discret. Math. 2 (1) (1972) 17–29.

[88] M. Sutton, T. Ben-Nun, A. Barak, Optimizing parallel graph connectivity computation via subgraph sampling, in: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2018, pp. 12–21.

[89] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (2) (1972) 146–160.

[90] E. Nuutila, E. Soisalon-Soininen, On finding the strongly connected components in a directed graph, Inf. Process. Lett. 49 (1) (1994) 9–14.

[91] S.E. Garza, S.E. Schaeffer, Community detection with the label propagation algorithm: a survey, Phys. a Stat. Mech. Appl. 534 (2019) 122058.

[92] J. Xie, B.K. Szymanski, X. Liu, Slpa: uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process, in: 2011 Ieee 11th International Conference on Data Mining Workshops, IEEE, 2011, pp. 344–349.

[93] I.X. Leung, P. Hui, P. Lio, J. Crowcroft, Towards real-time community detection in large networks, Physical Review E—Statistical, Nonlinear, and Soft Matter Physics 79 (6) (2009) 066107.

[94] M.E. Newman, M. Girvan, Finding and evaluating community structure in networks, Phys. Rev. E 69 (2) (2004) 026113.

[95] X.-S. Zhang, R.-S. Wang, Y. Wang, J. Wang, Y. Qiu, L. Wang, L. Chen, Modularity optimization in community detection of complex networks, Europhysics Letters 87 (3) (2009) 38002.

[96] V.D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, Journal of statistical mechanics: theory and experiment 2008 10 (2008) 10008.

[97] H. Lu, M. Halappanavar, A. Kalyanaraman, Parallel heuristics for scalable community detection, Parallel Comput. 47 (2015) 19–37.

[98] K.P. Sinaga, M.-S. Yang, Unsupervised k-means clustering algorithm, IEEE Access 8 (2020) 80716–80727.

[99] C.W. Commander, Maximum CUT problem, Max-cut, in: Encyclopedia of Optimization, vol. 2, Springer US, 2008, pp. 1991–1999.

[100] P. Festa, P.M. Pardalos, M.G. Resende, C.C. Ribeiro, Randomized heuristics for the max-cut problem, Optim. Methods Softw. 17 (6) (2002) 1033–1058.

[101] U.V. Luxburg, A tutorial on spectral clustering, Stat. Comput. 17 (4) (2007) 395–416.

[102] M. Tan, S. Zhang, L. Wu, Mutual KNN based spectral clustering, Neural Comput. Appl. 32 (11) (2020) 6435–6442.

[103] Y. Fang, K. Wang, X. Lin, W. Zhang, Cohesive subgraph search over big heterogeneous information networks: applications, challenges, and solutions, in: Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 2829–2838.

[104] F. Zhao, A.K. Tung, Large scale cohesive subgraphs discovery for social network visual analysis, Proc. VLDB Endow. 6 (2) (2012) 85–96.

[105] F. Rousseau, M. Vazirgiannis, Main core retention on graph-of-words for single-document keyword extraction, in: European Conference on Information Retrieval, Springer, 2015, pp. 382–393.

[106] S.B. Seidman, Network structure and minimum degree, Soc. Netw. 5 (3) (1983) 269–287.

[107] H. Kabir, K. Madduri, Parallel k-core decomposition on multicore platforms, in: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2017, pp. 1482–1491.

[108] A. Tripathy, F. Hohman, D.H. Chau, O. Green, Scalable k-core decomposition for static graphs using a dynamic graph data structure, in: 2018 IEEE International Conference on Big Data (Big Data), IEEE, 2018, pp. 1134–1141.

[109] R.E. Tarjan, U. Vishkin, An efficient parallel biconnectivity algorithm, SIAM J. Comput. 14 (4) (1985) 862–874.

[110] H.N. Gabow, Path-based depth-rst search for strong and biconnected components, Inf. Process. Lett. 74 (3–4) (2000) 107–114.

[111] M. Chaitanya, K. Kothapalli, Efficient multicore algorithms for identifying biconnected components, International Journal of Networking and Computing 6 (1) (2016) 87–106.

[112] J.A. Ferland, G. Guénette, Decision support system for the school districting problem, Oper. Res. 38 (1) (1990) 15–21.

[113] H. Chen, T. Cheng, X. Ye, Designing efficient and balanced police patrol districts on an urban street network, Int. J. Geogr. Inf. Sci. 33 (2) (2019) 269–290.

[114] T. Mostafaie, F.M. Khiyabani, N.J. Navimipour, A systematic study on meta-heuristic approaches for solving the graph coloring problem, Comput. Oper. Res. 120 (2020) 104850.

[115] Ü.V. Çatalyürek, J. Feo, A.H. Gebremedhin, M. Halappanavar, A. Pothen, Graph coloring algorithms for multi-core and massively multithreaded architectures, Parallel Comput. 38 (10–11) (2012) 576–594.

[116] O. Titiloye, A. Crispin, Graph coloring with a distributed hybrid quantum annealing algorithm, in: Agent and Multi-Agent Systems: Technologies and Applications: 5th KES International Conference, KES-AMSTA 2011, Springer, Manchester, UK, June 29–July 1, 2011. Proceedings 5, 2011, pp. 553–562.
[117] D.S. Johnson, M. Yannakakis, C.H. Papadimitriou, On generating all maximal independent sets, Inf. Process. Lett. 27 (3) (1988) 119–123.
[118] S. Basagni, Finding a maximal weighted independent set in wireless networks, Telecommun. Syst. 18 (2001) 155–168.
[119] M. Krivelevich, T. Mészáros, P. Michaeli, C. Shikhelman, Greedy maximal independent sets via local limits, Random Structures & Algorithms 64 (4) (2024) 986–1015.
[120] J. Edmonds, Paths, trees, and flowers, Can. J. Math. 17 (1965) 449–467.
[121] V. Kolmogorov, Blossom v: a new implementation of a minimum cost perfect matching algorithm, Math. Program. Comput. 1 (1) (2009) 43–67.
[122] S. Bouhenni, S. Yahiaoui, N. Nouali-Taboudjemat, H. Kheddouci, A survey on distributed graph pattern matching in massive graphs, ACM Computing Surveys (CSUR) 54 (2) (2021) 1–35.
[123] P. Vilakone, D.-S. Park, K. Xinchang, F. Hao, An efficient movie recommendation algorithm based on improved k-clique, Hum.-centric Comput. Inf. Sci. 8 (2018) 1–15.
[124] L. Becchetti, P. Boldi, C. Castillo, A. Gionis, Efficient semi-streaming algorithms for local triangle counting in massive graphs, in: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2008, pp. 16–24.
[125] Y.-J. Chang, S. Pettie, T. Saranurak, H. Zhang, Near-optimal distributed triangle enumeration via expander decompositions, Journal of the ACM (JACM) 68 (3) (2021) 1–36.
[126] Y.-J. Chang, T. Saranurak, Improved distributed expander decomposition and nearly optimal triangle enumeration, in: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, 2019, pp. 66–73.
[127] S. Ghosh, Improved distributed-memory triangle counting by exploiting the graph structure, in: 2022 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, 2022, pp. 1–6.
[128] S. Pandey, Z. Wang, S. Zhong, C. Tian, B. Zheng, X. Li, L. Li, A. Hoisie, C. Ding, D. Li, et al., Trust: triangle counting reloaded on gpus, IEEE Trans. Parallel Distrib. Syst. 32 (11) (2021) 2646–2660.
[129] L. Becchetti, P. Boldi, C. Castillo, A. Gionis, Efficient algorithms for large-scale local triangle counting, ACM Transactions on Knowledge Discovery from Data (TKDD) 4 (3) (2010) 1–28.
[130] R.D. Luce, A.D. Perry, A method of matrix analysis of group structure, Psychometrika 14 (2) (1949) 95–116.
[131] J.J. McGregor, Backtrack search algorithms and the maximal common subgraph problem, Softw. Pract. Exp. 12 (1) (1982) 23–34.
[132] Y. Cao, T. Jiang, T. Girke, A maximum common substructure-based algorithm for searching and predicting drug-like compounds, Bioinformatics 24 (13) (2008) i366–i374.
[133] N. Shibata, Y. Kajikawa, I. Sakata, Link prediction in citation networks, J. Am. Soc. Inf. Sci. Technol. 63 (1) (2012) 78–85.
[134] H. Ma, On measuring social friend interest similarities in recommender systems, in: Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval, 2014, pp. 465–474.
[135] L. Yao, L. Wang, L. Pan, K. Yao, Link prediction based on common-neighbors for dynamic social network, Procedia Comput. Sci. 83 (2016) 82–89.
[136] T. Cover, Estimation by the nearest neighbor rule, IEEE Trans. Inf. Theory 14 (1) (1968) 50–55.
[137] L.E. Peterson, K-nearest neighbor, Scholarpedia 4 (2) (2009) 1883.
[138] W. Dong, C. Moses, K. Li, Efficient k-nearest neighbor graph construction for generic similarity measures, in: Proceedings of the 20th International Conference on World Wide Web, 2011, pp. 577–586.
[139] P. Jaccard, Étude comparative de LA distribution florale dans une portion des alpes ET des Jura, Bull Soc Vaudoise Sci Nat 37 (1901) 547–579.
[140] L.A. Adamic, E. Adar, Friends and neighbors on the web, Soc. Netw. 25 (3) (2003) 211–230.
[141] T. Murata, S. Moriyasu, Link prediction of social networks based on weighted proximity measures, in: IEEE/WIC/ACM International Conference on Web Intelligence (WI'07), IEEE, 2007, pp. 85–88.
[142] T. Zhou, L. Lu, Y.-C. Zhang, Predicting missing links via local information, Eur. Phys. J. B 71 (2009) 623–630.
[143] S. Peyer, D. Rautenbach, J. Vygen, A generalization of dijkstra's shortest path algorithm with applications to VLSI routing, J. Discrete Algorithms 7 (4) (2009) 377–390.
[144] E.F. Moore, The shortest path through a maze, in: Proc. Of the International Symposium on the Theory of Switching, Harvard University Press, 1959, pp. 285–292.
[145] L. Luo, M. Wong, W.-M. Hwu, An effective GPU implementation of breadth-first search, in: Proceedings of the 47th Design Automation Conference, 2010, pp. 52–55.
[146] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, 2022.
[147] E.W. Dijkstra, A note on two problems in connexion with graphs, in: Edsger Wybe Dijkstra: His Life, Work, and Legacy, Association for Computing Machinery (ACM) 2022, pp. 287–290.
[148] U. Meyer, P. Sanders, δ-stepping: a parallelizable shortest path algorithm, J. Algorithms 49 (1) (2003) 114–152.
[149] L.R. Ford, Network flow theory, Rand Corporation Paper, Santa Monica, pp. 1956, 1956.
[150] R. Bellman, On a routing problem, Q. Appl. Math. 16 (1) (1958) 87–90.
[151] Y. Han, T. Takaoka, An $o(n^3 \text{ loglog } n/\log^2 n)$ time algorithm for all pairs shortest paths, J. Discrete Algorithms 38 (2016) 9–19.
[152] J.Y. Yen, An algorithm for finding shortest routes from all source nodes to a given destination in general networks, Q. Appl. Math. 27 (4) (1970) 526–530.
[153] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. 4 (2) (1968) 100–107.
[154] K. Riesen, H. Bunke, Approximate graph edit distance computation by means of bipartite graph matching, Image Vis. Comput. 27 (7) (2009) 950–959.
[155] D.B. Blumenthal, S. Bougleux, J. Gamper, L. Brun, Gedlib: a c++ library for graph edit distance computation, in: International Workshop on Graph-Based Representations in Pattern Recognition, Springer, 2019, pp. 14–24.
[156] R.C. Prim, Shortest connection networks and some generalizations, The Bell System Technical Journal 36 (6) (1957) 1389–1401.
[157] M.P. de Aragao, R.F. Werneck, On the implementation of mst-based heuristics for the steiner problem in graphs, in: Workshop on Algorithm Engineering and Experimentation, Springer, 2002, pp. 1–15.
[158] R.C. Rocha, B.D. Thatte, Distributed cycle detection in large-scale sparse graphs, in: Proceedings of Simpósio Brasileiro De Pesquisa Operacional (SBPO'15), 2015, pp. 1–11.
[159] L.R. Ford Jr, D.R. Fulkerson, A simple algorithm for finding maximal network flows and an application to the hitchcock problem, Can. J. Math. 9 (1957) 210–218.
[160] M.T. Kyi, L.L. Naing, Application of ford-fulkerson algorithm to maximum flow in water distribution pipeline network, Int. J. Sci. Res. Publ. 8 (12) (2018) 306–310.
[161] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 2010, pp. 135–146.
[162] C. Avery, Giraph: large-scale graph processing infrastructure on hadoop, Proceedings of the Hadoop Summit. 11 (3) (2011) 5–9 Santa Clara.
[163] G. Csardi, T. Nepusz, The igraph software, Complex Syst. 1695 (2006) 1–9.
[164] T.P. Peixoto, The graph-tool Python library, 2017, https://graph-tool.skewed.de/.
[165] Neo4j, 2026, Retrieved from https://neo4j.com/ (26 January 2026).
[166] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, A. Taylor, Cypher: an evolving query language for property graphs, in: Proceedings of the 2018 International Conference on Management of Data, 2018, pp. 1433–1445.
[167] A. Arrow, Apache arrow, 2026, Retrieved from https://arrow.apache.org/ (26 January 2026).
[168] A. C. DB, Azure cosmos db, 2026, Retrieved from https://azure.microsoft.com/en-us/products/cosmos-db (26 January 2026).
[169] M.A. Rodriguez, The gremlin graph traversal machine and language (invited talk), in: Proceedings of the 15th Symposium on Database Programming Languages, 2015, pp. 1–10.
[170] GraphFrames, 2026, Retrieved from https://graphframes.io/ (26 January 2026).
[171] Apache spark, 2026, Retrieved from https://spark.apache.org/ (26 January 2026).
[172] Arango DB, 2026, Retrieved from https://arangodb.com/ (26 January 2026).
[173] AQL, 2026, Retrieved from https://docs.arangodb.com/stable/aql/ (26 January 2026).
[174] OrientDB, 2026, Retrieved from https://orientdb.org/ (26 January 2026).
[175] Amazon neptune, 2026, Retrieved from https://aws.amazon.com/it/neptune/ (26 January 2026).
[176] NebulaGraph, 2026, Retrieved from https://www.nebula-graph.io/ (26 January 2026).
[177] nGQL, 2026, Retrieved from https://docs.nebula-graph.io/3.8.0/ (26 January 2026).
[178] GraphX, 2026, Retrieved from https://spark.apache.org/graphx/ (26 January 2026).
[179] Memgraph, 2026, Retrieved from https://memgraph.com/docs (26 January 2026).
[180] Apache Hadoop, 2026, Retrieved from https://hadoop.apache.org/ (26 January 2026).
[181] Apache TinkerPop, 2026, Retrieved from https://tinkerpop.apache.org/ (26 January 2026).
[182] TigerGraph, 2026, Retrieved from https://www.tigergraph.com/ (26 January 2026).
[183] GSQL, 2026, Retrieved from https://docs.tigergraph.com/gsql-ref/4.2/intro/ (26 January 2026).
[184] Dgraph, 2026, Retrieved from https://dgraph.io/ (26 January 2026).
[185] DQL, 2026, Retrieved from https://dgraph.io/docs/dql/ (26 January 2026).
[186] Ultipa, 2026, Retrieved from https://ultipa.com/ (26 January 2026).
[187] Oracle, Oracle graph database, 2026, Retrieved from https://www.oracle.com/database/integrated-graph-database/ (26 January 2026).
[188] Kùzu, 2026, Retrieved from https://github.com/kuzudb/kuzu (26 January 2026).
[189] Sparksee, 2026, Retrieved from www.sparsity-technologies.com/sparksee-graph-database-management-system/ (26 January 2026).
[190] VelocityDB, 2026, Retrieved from https://velocitydb.com/ (26 January 2026).
[191] DataStax, Datastax graph, 2026, Retrieved from https://docs.datastax.com/en/dse/6.9/graph/graph-contents.html (26 January 2026).
[192] HGraphDB, 2026, Retrieved from https://github.com/rayokota/hgraphdb?tab=readme-ov-file (26 January 2026).
[193] AgensGraph, 2026, Retrieved from https://bitnine.net/agensgraph/?ckattempt=1 (26 January 2026).
[194] D. de Graaf, Algorithm support in a graph database, done right, in: Proceedings of the VLDB 2025 PhD Workshop, vol. 2150-8097, 2025.

[195] F. Cambria, Repository with synthetic graph datasets for algorithm evaluation: random, scale-free, and small-world distributions, 2026, https://doi.org/10.5281/zenodo.19252778
[196] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, J. Leskovec, Open graph benchmark: datasets for machine learning on graphs, Adv. Neural Inf. Process. Syst. 33 (2020) 22118–22133.
[197] ArXiv, 2026, Retrieved from https://arxiv.org/ (26 January 2026).
[198] D.S. Wishart, Y.D. Feunang, A.C. Guo, E.J. Lo, A. Marcu, J.R. Grant, T. Sajed, D. Johnson, C. Li, Z. Sayeeda, et al., Drugbank 5.0: a major update to the drugbank database for 2018, Nucleic acids research 46 (D1) (2018) D1074–D1082.
[199] Z. Wu, B. Ramsundar, E.N. Feinberg, J. Gomes, C. Geniesse, A.S. Pappu, K. Leswing, V. Pande, Moleculenet: a benchmark for molecular machine learning, Chem. Sci. 9 (2) (2018) 513–530.