Revisiting Graph Analytics Benchmark

LINGKAI MENG[*], Antai College of Economics and Management, Shanghai Jiao Tong University, China
YU SHAO[*], East China Normal University, China
LONG YUAN[†], Wuhan University of Technology, China
LONGBIN LAI, Alibaba Group, China
PENG CHENG, Tongji University, China
XUE LI, Alibaba Group, China
WENYUAN YU, Alibaba Group, China
WENJIE ZHANG, University of New South Wales, Australia
XUEMIN LIN, Antai College of Economics and Management, Shanghai Jiao Tong University, China
JINGREN ZHOU, Alibaba Group, China

The rise of graph analytics platforms has led to the development of various benchmarks for evaluating and comparing platform performance. However, existing benchmarks often fall short of fully assessing performance due to limitations in core algorithm selection, data generation processes (and the corresponding synthetic datasets), as well as the neglect of API usability evaluation. To address these shortcomings, we propose a novel graph analytics benchmark. First, we select eight core algorithms by extensively reviewing both academic and industrial settings. Second, we design an efficient and flexible data generator and produce eight new synthetic datasets as the default datasets for our benchmark. Lastly, we introduce a multi-level large language model (LLM)-based framework for API usability evaluation-the first of its kind in graph analytics benchmarks. We conduct comprehensive experimental evaluations on existing platforms (GraphX, PowerGraph, Flash, Grape, Pregel+, Ligra, and G-thinker). The experimental results demonstrate the superiority of our proposed benchmark.

CCS Concepts: • **Information systems → Database performance evaluation**.

Additional Key Words and Phrases: Graph Analytics Benchmarks, Data Generator, API Usability Evaluation

ACM Reference Format:
Lingkai Meng, Yu Shao, Long Yuan, Longbin Lai, Peng Cheng, Xue Li, Wenyuan Yu, Wenjie Zhang, Xuemin Lin, and Jingren Zhou. 2025. Revisiting Graph Analytics Benchmark. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 208 (June 2025), 28 pages. https://doi.org/10.1145/3725345

[*]These authors contributed equally to this work and are listed in alphabetical order by last name.
[†]Corresponding author.

Authors' Contact Information: Lingkai Meng, Antai College of Economics and Management, Shanghai Jiao Tong University, Shanghai, China, mlk123@sjtu.edu.cn; Yu Shao, East China Normal University, Shanghai, China, yushao@stu.ecnu.edu.cn; Long Yuan, Wuhan University of Technology, Wuhan, China, longyuanwhut@gmail.com; Longbin Lai, Alibaba Group, Hangzhou, China, longbin.lailb@alibaba-inc.com; Peng Cheng, Tongji University, Shanghai, China, cspcheng@tongji.edu.cn; Xue Li, Alibaba Group, Hangzhou, China, youli.lx@alibaba-inc.com; Wenyuan Yu, Alibaba Group, Hangzhou, China, wenyuan.ywy@alibaba-inc.com; Wenjie Zhang, University of New South Wales, Sydney, Australia, wenjie.zhang@unsw.edu.au; Xuemin Lin, Antai College of Economics and Management, Shanghai Jiao Tong University, Shanghai, China, xuemin.lin@gmail.com; Jingren Zhou, Alibaba Group, Hangzhou, China, jingren.zhou@alibaba-inc.com.

ACM 2836-6573/2025/6-ART208
https://doi.org/10.1145/3725345

1 Introduction

Graph analytics platforms, such as GraphX [23], PowerGraph [22], Flash [40], Grape [17], Pregel+ [44, 92, 93], Ligra [72], and G-thinker [94], have been developed to facilitate large-scale graph data analysis. These platforms vary significantly in their design principles, architectures, and implementation details, which raises an important and practical question: how to evaluate graph analytics platforms and provide guidance on selecting the most appropriate one for specific environments.

Accordingly, graph analytics benchmarks like LDBC Graphalytics [28] and Graph500 [24] are commonly used to evaluate platform performance. These benchmarks provide a standardized framework for assessment, typically consisting of two main components: core algorithms and datasets. Core algorithms measure the efficiency of various graph processing tasks, offering a consistent basis for comparing platforms. Datasets aim to represent diverse graph characteristics found in real-world applications. However, due to high collection costs and privacy concerns, real-world datasets are often limited. As a result, benchmarks frequently use data generators to produce synthetic datasets that mimic real-world graphs.

Motivation. Considering the goals and key components of graph analytics benchmarks, an ideal benchmark should select the minimal number of algorithms while ensuring they are representative and diverse enough to effectively distinguish the performance of various graph analytics platforms. The data generator should also efficiently produce synthetic datasets, allowing users to control input parameters to generate graphs with diverse characteristics. Unfortunately, existing benchmarks fall short of these ideals. Take the most popular LDBC Graphalytics benchmark [28] as an example. Regarding core algorithms, LDBC Graphalytics selects PageRank (PR) [61], Breadth First Search (BFS) [36], Single Source Shortest Path (SSSP) [12], Weakly Connected Component (WCC) [75], Label Propagation Algorithm (LPA) [66], Local Clustering Coefficient (LCC) [27] as its core algorithms. However, as analyzed in Section 3, these six algorithms lack diversity, with most having linear time complexity, limiting their ability to reveal platform performance bottlenecks. Regarding dataset generation, LDBC Graphalytics's data generator employs a sampling-based strategy to generate datasets, which, as examined in Section 4, suffers from efficiency issues due to numerous failed trials. Additionally, the generator only allows control over the scale of datasets, lacking flexibility in mimicking real-world datasets of different characteristics.

Moreover, current benchmarks predominantly assess and compare performance metrics such as execution time, throughput, and scalability. However, these benchmarks often overlook the usability of the application programming interfaces (APIs) provided by graph analytics platforms. APIs play a crucial role in enhancing developer productivity, minimizing errors, and fostering wider platform adoption. The success of a graph analytics platform is significantly tied to the usability of its APIs, as highlighted in studies [56, 68]. Consequently, API usability should be a key component in the design of graph analytics benchmarks. Yet, existing benchmarks typically neglect API usability, relying instead on simplistic measures like the number of lines of code to implement tasks [58, 85, 97]. Such metrics are insufficient for a comprehensive evaluation of API usability.

Motivated by these gaps, in this paper, we revisit the problem of graph analytics benchmarks and aim to develop a new benchmark that includes a diverse set of core algorithms, an efficient and flexible data generator, and a rigorous method for evaluating API usability.

Our Approach. We address the three major limitations in existing graph analytics benchmarks. For the core algorithms, we conduct an extensive review of both academic research and industrial applications, selecting eight algorithms (PR, SSSP, LPA, WCC, Triangle Counting (TC), Betweenness Centrality (BC), k-Clique (KC), Core Decomposition (CD)) based on their popularity, time complexity, and diversity of algorithmic topics. For the data generator, we redesign the edge sampling strategy and introduce a failure-free generator that eliminates the inefficiencies caused by

Table 1. Comparison of benchmarks for graph analytics platforms

Benchmarks	Core Algorithms	Synthetic Datasets	Evaluation Metrics	
			Performance	Usability
Graph500 [24]	BFS, SSSP	Scale	Timing, Throughput	–
WGB [4]	K-Hop [83], SSSP, PR, WCC, Cluster [71]	Scale, Density	Timing, Scalability	–
BigDataBench [82]	BFS, PR, WCC, Cluster	Scale	Timing, Throughput, MIPS, cache MPKI	–
LDBC Graphalytics [28]	PR, BFS, SSSP, WCC, LPA, LCC	Scale	Timing, Throughput, Scalability, Robustness	–
Ours	PR, SSSP, TC, BC, KC, CD, LPA, WCC	Scale, Density, Diameter	Timing, Throughput, Scalability, Robustness	LLM-based

failed trials in LDBC Graphalytics. Additionally, recognizing the critical impact of graph density and diameter on platform performance [17, 40, 92], we develop a mechanism that allows users to control these two attributes of the generated datasets, significantly enhancing the flexibility of the data generator. For API usability evaluation, we harness the capabilities of large language models (LLMs) and propose a multi-level LLM-based evaluation framework that simulates programmers of varying expertise levels, providing a comprehensive assessment of API usability. Table 1 compares our proposed benchmark with existing graph analytics benchmarks regarding core algorithms, controllable attributes of the synthetic datasets, and the evaluation metrics.

Contributions. In this paper, we make the following contributions:

- We revisit existing graph analytics benchmarks, identify their limitations, and introduce a more comprehensive benchmark that supports a broader range of algorithms and datasets, along with a richer set of evaluations, including the assessment of API usability.
- We select eight representative core algorithms, spanning a wide range of algorithm popularity, algorithm diversity and computing models, to ensure comprehensive and realistic evaluation of platform performance (Section 3).
- We design an efficient and flexible data generator and produce eight synthetic datasets using our new data generator that serve as the default for our benchmark (Section 4).
- We propose a LLM-based evaluation framework to assess the API usability. To the best of our knowledge, this is the first graph analytics benchmark that includes a rigorous method for evaluating API usability (Section 5).
- We conduct a comprehensive experimental evaluation on existing graph analytics platforms using the newly proposed benchmark, and the experimental results demonstrate the superiority of our benchmark (Section 8).

2 Related Work

Benchmarks for Graph Analytics Platforms. Benchmarking graph analytics platforms has attracted lots of attention in recent years, and many general-purpose benchmarks have been proposed in the literature. Graph500 [24] uses BFS and SSSP algorithms on synthetic datasets generated by the Kronecker graph generator to measure platform performance in terms of timing and throughput. WGB [4] provides an efficient data generator for creating dynamic graphs similar to real-world graphs, offering a universal benchmark for graph analytics platforms. BigDataBench [82] aims to provide a comprehensive benchmarking suite for data systems, encompassing various data types, including graph data, text, and tables, and focusing on performance, energy efficiency, and cost-effectiveness under diverse workloads and data scenarios. LDBC [28] is the most popular benchmark for graph analysis systems, offering datasets and evaluation metrics for various tasks, establishing itself as the state-of-the-art tool for assessing graph analytics platforms.

Synthetic Graph Data Generators in Benchmarks. Graph data generator creates synthetic graph data that simulates real-world network structures and behaviors for use in benchmarking and analysis. Traditional Erdős-Rényi generator [13] iteratively selects two random vertices to generate an edge. However, the generated graph has a low clustering coefficient and follows a Poisson degree distribution, which is contrary to real-world networks. Two improved generators, Watts-Strogatz [86] and Barabási-Albert [6], can generate highly-clustered and power-law graphs, respectively, while the Kronecker [37] generator used in Graph500 [54] can simultaneously follow these two properties. The LFR3 [35] generator allows for the tuning of parameters to control clusters and power-law distribution. The LDBC-DG generator can further generate and fit arbitrary degree distribution and has a better community similarity [14]. In addition to the general graph generator, the WGB benchmark [4] adds a dynamic graph generator for evaluating systems under dynamic conditions. Many generators are integrated in graph libraries such as NetworkX [26] and Neo4j-APOC [2]. However, these generators are not efficient enough and cannot control the density and diameter of the generated graphs.
API Usability Evaluation. Due to the importance of APIs in software development, many studies [18, 25, 52, 56, 62, 68] have explored methods for evaluating API usability to guide API design. Existing works mainly rely on expert evaluations by project managers, developers, and testers, who review the API design and documentation to evaluate its usability and suggest improvements [18, 68], while others gather insights through long-term tracking of developer feedback [53]. Brad A. Myers [56] points out that better API usability requires designing APIs to effectively meet the needs of users at different levels, including novices, professionals, and end-user programmers. Therefore, some studies [56, 62] have adopted the method of collecting feedback from users with varying levels of expertise to evaluate API usability. Clearly, this approach is costly and difficult to scale for large benchmark testing.

3 Core Algorithms

Algorithms play a vital role in benchmarking distributed graph processing platforms. LDBC [28], the most popular benchmark for graph processing platforms, selects six algorithms (BFS, PR, WCC, LPA, LCC and SSSP) frequently mentioned in academic papers as its core algorithms. However, the LDBC's algorithm set lacks diversity, as most of the algorithms focus on community detection and traversal, ignoring the suitability to different computing models.

To overcome these limitations, we consider multiple criteria when selecting core algorithms to ensure a comprehensive and representative benchmark including (1) popularity in academic research and real-world application, (2) diversity in algorithm topics, and (3) suitability for distributed graph computing models. Based on these criteria, we select eight algorithms, including:

- **PageRank** (PR) measures the importance of each vertex based on the number and quality of its edges.
- **Label Propagation Algorithm** (LPA) finds communities by exchanging labels between vertices, using a semi-supervised approach.
- **Single Source Shortest Path** (SSSP) finds the shortest paths from a given source vertex to all others based on the smallest sum of edge weights.
- **Weakly Connected Component** (WCC) identifies subgraphs where any two vertices are connected, ignoring edge direction.
- **Betweenness Centrality** (BC) quantifies the degree to which a vertex lies on the shortest paths between pairs of other vertices.
- **Core Decomposition** (CD) determines the coreness value of each vertex, indicating if it is part of a k-Core subgraph, where it is connected to at least k other vertices.

Table 2. Popularity of selected core algorithms

Algorithms	#Papers	DBLP	Google Scholar	WoS
PR	28	1012	25400	4554
LPA	39	771	130000	1195
SSSP	33	584	17800	2252
WCC	26	835	17800	726658
BC	20	304	43900	5634
CD	29	179	126000	19499
TC	27	252	20500	1784
KC	31	352	41800	395

Table 3. Workload and topics

Algorithms	Workload	Topic	LDBC	Ours
PR	$O(k \cdot m)$	Centrality	✓	✓
LPA	$O(k \cdot m)$	Community Detection	✓	✓
SSSP	$O(m + n \cdot \log n)$	Traversal	✓	✓
WCC	$O(m + n)$	Community Detection	✓	✓
BC	$O(n^3)$	Centrality		✓
CD	$O(m + n)$	Cohesive Subgraph		✓
TC	$O(m^{1.5})$	Pattern Matching		✓
KC	$O(k^2 \cdot n^k)$	Pattern Matching		✓
BFS	$O(m + n)$	Traversal	✓	
LCC	$O(m^{1.5})$	Community Detection	✓	

- **Triangle Counting (**TC**)** counts the number of triangles in a graph, often used to measure the local clustering coefficient.
- ***k*-Clique (**KC**)** identifies all complete subgraphs with k vertices.

3.1 Algorithm Popularity

Table 2 presents statistics on the number of papers related to the selected algorithms, appearing in representative conferences and journals [51]. It also shows their frequency of appearance in research engines like DBLP, Google Scholar, and Web of Science (WoS) over the past ten years. Moreover, the selected algorithms are not only frequently studied in academic papers but also have a wide range of practical applications. Here are just a few examples: **(1) Social Network Analysis.** PR and BC are used to detect important individuals in social networks by considering the influence of neighbors [21, 31, 96] and shortest paths [33], respectively. LPA and KC are used to detect communities [5, 60], while CD can further reveal the hierarchical structure [47]. **(2) Computer Vision.** WCC can find connected pixels for unsupervised segmentation [77, 81]. LPA can utilize given labels for semantic segmentation [55, 79]. **(3) Biological Analysis.** Similar to the social network analysis, BC and PR are also adaptive to measure the importance of proteins [29, 84], while TC can measure the similarity like Jaccard [8]. In epidemiology, CD is suitable to reveal and predict epidemic outbreaks. **(4) Road Network Routing.** On road network, SSSP and WCC are used to compute shortest paths [20, 59] and check the connection [87, 89] between vertices, respectively. BC is used to detect frequent area that most shortest paths passed [32, 39]. **(5) Recommendation Systems.** PR can detect high influence items for recommendation [45]. TC and KC can detect clique and triadic instance to mine a close relation for further recommendation [9, 10, 88]. KC and LPA can also detect fraud information [7, 42].

3.2 Algorithm Diversity

Table 3 further compares the computational workload and topics of our core algorithms and LDBC. Our algorithms span a range of complexities, from linear (PR, CD, WCC, LPA) to logarithmic-linear (SSSP), polynomial (TC, BC), and even higher-order complexities (KC), providing a balanced coverage of different workload levels. In contrast, more than half of the algorithms (PR, LPA, WCC, BFS) in LDBC are linear. Regarding the topics, our core algorithms cover areas including centrality, community detection, traversal, cohesive subgraph, and pattern matching, highlighting the diversity of our algorithms, while LDBC only includes centrality, community detection, traversal, with half of its algorithms focused on community detection. Overall, our core algorithms are widely used in academia and industry, addressing real-world needs while covering diverse workloads, enabling our benchmark to identify platform bottlenecks and strengths effectively.

3.3 Algorithm Suitability for Computing Models

The performance of distributed graph algorithms is significantly impacted by the underlying computing model. In the following, we explore common graph computing models and showcase the varying suitability of the selected algorithms across these models.

Graph Computing Models. The Bulk Synchronous Parallelism (BSP) model, introduced by Valiant [78], is a popular computing model for parallel graph computation. It divides the computation into multiple supersteps, where each superstep consists of three phases: local computation, communication, and synchronization, for efficiently executing parallel tasks.

On top of the BSP model, the vertex-centric model was developed with the "Think-Like-a-Vertex" (TLAV) philosophy [49]. In vertex-centric model, the vertex is the basic unit of computing and scheduling. Each vertex can process computation locally and send messages to other vertices. Users are only required to implement a vertex-based computation interface, while the platform will execute it over all vertices for a certain number of iterations or until it converges. There are a few variants of vertex-centric model such as Single-Phase (Pregel [46], Pregel+ [92], Flash [41], Ligra [73]), Scatter-Gather (Signal/Collect [74]), and Scatter-Combine (GRE [93]). Specially, Spark also supports vertex-centric by providing Pregel-like interfaces, i.e., VertexRDD in GraphX [23]. Vertex-centric models face potential challenges, including load imbalance, which becomes particularly problematic in power-law graphs, and significant communication overhead when deployed on large-scale clusters.

Some platforms build upon the vertex-centric model by extending computation on edges or groups of vertices, leading to the development of edge-centric and block-centric models, respectively. PowerGraph [22], X-Stream [70], Graphchi [34] and Chaos [69] provide an edge-centric model to execute tasks over edges to resolve load skew in power-law graphs and fully utilize sequential reading and writing of solid-slate disk. Alternatively, Blogel [91] and Grape [15] are block-centric platforms that divide the graph into multiple blocks, such that vertex functions can communicate without communication if they belong to the same block. However, edge-centric does not support non-neighbor communication, while it's often non-trivial to program with the block-centric model.

All the above computing models produce output sizes proportional to the graph size, making them unsuitable for graph-mining problems that may yield an exponential number of results [90, 95]. To address this, platforms like Arabesque [76], Fractal [11], AutoMine [48], Peregrine [30], and G-thinker [94] adopt the subgraph-centric model. In this model, the fundamental computing unit is the subgraph (e.g., a triangle) rather than vertices, vertex groups, or edges. Users need to define how to construct candidate subgraphs and implement a computation interface for each subgraph.

Algorithm Categorizations. Different graph computing models can greatly influence algorithm efficiency, as their diverse algorithmic requirements may align differently with the characteristics of these models. Drawing on insights from prior studies [40, 91], we categorize the eight selected algorithms into three main classes and analyze their suitability across different computing models.

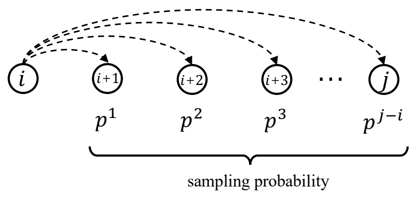


Fig. 1. Probabilities distribution used in LDBC-DG

- **Iterative Algorithms (**PR, LPA**)** involve multiple iterations, where each vertex is updated based on its neighbors' statuses, continuing until convergence or a fixed number of iterations is reached. The vertex-centric model is well-suited for iterative algorithms as it applies the update function directly to each vertex. These algorithms can also be efficiently implemented using the edge-centric model, particularly for handling highly skewed graphs.
- **Sequential Algorithms (**SSSP, WCC, BC, CD**)** require a specific execution order and often involve frequent synchronizations, such as iterations in SSSP. The block-centric model is particularly well-suited for such algorithms, especially those with fewer iterations. While it is possible to implement algorithms using the vertex-centric and edge-centric models, this can lead to suboptimal performance. However, some vertex-centric-based platforms [41, 73] can execute computation on a vertex subset instead of activate all vertices to minimize unnecessary computations.
- **Subgraph Algorithms (**TC, KC**)** involve iterative subgraph matching starting from a set of vertices, requiring intensive communication and computation. The vertex-centric model struggles with handling the high communication overhead in such cases. While the edge-centric model can do simpler tasks like TC, where only one edge and its two endpoints are needed to count triangles, it becomes inadequate for more complex subgraphs, such as cliques. In contrast, the subgraph-centric model is specifically designed for graph mining problems, enabling the generation of multiple subgraphs and the definition of computation functions directly based on subgraphs, making it well-suited for these tasks.

4 Data Generator & Synthetic Datasets

Benchmarking large-scale platforms requires datasets with diverse statistics. However, real-world datasets are often limited due to high collection costs and conflict-of-interest concerns. As a result, synthetic graph datasets have emerged as a practical alternative, generated by data generators that take specific parameters to produce the desired graphs.

Among various data generators, the LDBC Graphalytics generator (LDBC-DG) is noted for its high-quality simulation of real-life social graphs [64]. Specifically, LDBC-DG (1) initially generates vertices with some properties (such as *location* and *interest*). (2) Then, vertices are sorted with some specific similarity metrics (e.g., Z-order for *location* and identifier order for *interest*). (3) Finally, edges are formed with probabilities influenced by the distances between vertices in this ordered sequence [14].

Briefly, LDBC-DG follow the "Homophily Principle" [50] that individuals with similar characteristics are more likely to form connections. Therefore, similar vertices are closer in distance and have a higher probability of being connected. LDBC-DG defines the edge formation probability between vertex u_i and u_j as follows: $\Pr[e(u_i, u_j)] = max\{p^{|j-i|}, p_{limit}\}$, where p is the base probability, i and j represents the position of u_i and u_j in the ordered sequence, p_{limit} is a lower-bound. By default, p and p_{limit} are 0.95 and 0.2, respectively. To generate edges, LDBC-DG iterates over each vertex u_i and progressively samples $u_j (j > i)$ with the probability $\Pr[e(u_i, u_j)]$ as shown in Figure 1 until the number of required edges is obtained. However, LDBC-DG has two key limitations:

- **Inflexibility:** LDBC-DG focuses on producing social networks of different scales. However, graphs of the same scale can differ in characteristics like density and diameter, affecting platform performance. For instance, subgraph algorithms are sensitive to graph density, while sequential algorithms can be impacted by diameter. Some platforms offer optimizations for high-density [17, 92] or large-diameter graphs [91]. To effectively evaluate platform performance, the data generator should provide the flexibility of creating graphs with varied characteristics.
- **Inefficiency:** LDBC-DG samples each edge successively and individually. However, the exponential function decays rapidly, leading to a low sampling probability and more unsuccessful attempts, which is time-consuming and inefficient especially when generating a sparse graph.

To overcome these limitations, we propose the Failure-Free Trial Data Generator (FFT-DG). Compared to LDBC-DG, FFT-DG can offer greater flexibility to control the density and diameter as well as better similarity to the real-world datasets, while the efficiency is also improved. Utilizing FFT-DG, we can create various datasets with differing scales, densities, and diameters, serving as the default synthetic datasets for our benchmark.

4.1 Failure-Free Trial Generator

Our FFT-DG also has three steps and the initial two steps are the same as LDBC-DG. In the final edge generation, instead, FFT-DG directly extracts existing edges, bypassing intermediate failed trials. This ensures that the number of trials matches the number of generated edges, significantly reducing unnecessary sampling overhead.

As shown in Figure 2, we introduce a new probability function. For a vertex u_i, the probability of connecting an edge to u_j $(j > i)$ is defined as $\frac{1}{c+(j-i)}$. $c \geq 0$ is an adjustable parameter, where a larger c can reduce the probability of each edge. In FFT-DG, c defaults to 0 such that the adjacent edge $e(u_i, u_{i+1})$ always exists. As verified in Section 8.1, this new function does not affect the quality of generated graphs compared to that of LDBC-DG.

Unlike LDBC-DG, where each edge is sequentially sampled until $e(u_i, u_j)$ exists, FFT-DG can directly obtain the first existing edge and avoid sampling intermediate edges from $e(u_i, u_{i+1})$ to $e(u_i, u_{j-1})$. Specifically, the probability of an edge $e(u_i, u_j)$ being the first existing edge of u_i is:

$$\begin{aligned}\Pr[e(i,j)] &= \left(1 - \frac{1}{c+1}\right) \cdot \left(1 - \frac{1}{c+2}\right) \cdots \frac{1}{c+(j-i)} \\ &= \frac{c}{c+(j-i-1)} - \frac{c}{c+(j-i)}\end{aligned} \tag{1}$$

Then, the probabilities for each edge to be the first existing edge are structured such that the sequence of probabilities, $1 - \frac{c}{c+1}$, $\frac{c}{c+1} - \frac{c}{c+2}$, etc., forms a continuous range that spans from 0 to 1. By treating these probabilities as distinct segments that collectively cover the interval $(0, 1]$, we can randomly select a floating-point number f from this interval $(0, 1]$ to represent an edge $e(i, j)$ by $j = i + \left\lfloor \left(\frac{1}{f} - 1\right) \cdot c \right\rfloor + 1$. Once an edge is sampled (e.g., $e(u_i, u_j)$), the following edges $e(u_i, u_k)$ $(k>j)$ exhibit the similar properties:

$$\Pr[e(i,k)] = \prod_{t=j+1-i}^{k-1-i} \left(1 - \frac{1}{c+t}\right) \cdot \frac{1}{c+(k-i)}$$

Introducing $c' = c + (j - i)$ simplifies the expression:

$$\Pr[e(i,k)] = \frac{c'}{c'+(k-j-1)} - \frac{c'}{c'+(k-j)}$$

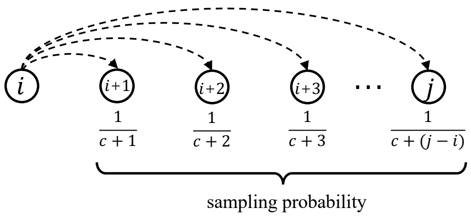


Fig. 2. Probabilities distribution used in FFT-DG

Algorithm 1: Failure-free Trial Data Generator

```
   Input: a vertices set $V$, the clustering parameter $\alpha$
   Output: the edges set $E$
1  sort $u_i \in V$ with a specific similarity metric
2  for $u_i \in V$ do
3      $c \leftarrow 0,\ j \leftarrow i$
4      while $u_i.degree < u_i.degree\_limit$ do
5          select $f$ from $(0, 1]$ randomly and uniformly
6          $k \leftarrow j + \left\lfloor \left(\frac{1}{f} - 1\right) \cdot \frac{c}{\alpha} \right\rfloor + 1$          ▷ Sampling
7          if $k > |V|$ then Break
8          if $u_k.degree < u_k.degree\_limit$ then
9              $E \leftarrow E \cup e(i, k)$          ▷ Generating
10         $c \leftarrow c + (k - j),\ j \leftarrow k$
11 return $E$
```

The probabilities for each edge to become the next existing edge are structured similarly as $1 - \frac{c'}{c'+1}$, $\frac{c'}{c'+1} - \frac{c'}{c'+2}$, and so forth. Consequently, we can consistently obtain connected edges by updating the parameter c without any failure samples. The details are outlined in Algorithm 1.

4.2 Flexibility Improvement

We present adjustments to FFT-DG (Algorithm 1) for generating graphs with varying densities and diameters. Note that density enhancement is feasible because of the proposed edge formation probability in Equation 1, while diameter adjustment can be integrated smoothly into LDBC-DG.

4.2.1 Density Enhancement. The value of the sampling functions both in LDBC-DG and our generator, i.e., $p^{|j-i|}$ and $\frac{1}{c+|j-i|}$, decreases rapidly and can only generate few edges.

To enhance the density and generate more edges, LDBC-DG adds a lower bound on the probability p_{limit}. However, this resolution is overly simplistic and lacks justification, as assigning the same connection probability p_{limit} to all distant vertices, regardless of their actual separation.

Density Factor. Our approach is adding a density factor $\alpha > 1$ (which is used to control the density of the generated graph) during the sampling process to enhance the generation of edges, as shown in Line 5 of Algorithm 1. The factor α serves to concentrate probability from distant to local vertices. More precisely, we can reformulate the probability of an edge being the first existing edge as follows: $\Pr[e(i, j)] = \frac{1}{c + \frac{d(d-1)}{c} + 2d - 1}$, where $d = j - i$ is the distance. The problem is the change when c is reduced to $\frac{c}{\alpha}$. An observation is that $\Pr[e(i, j)]$ has a pair of same values when $c = \frac{d(d-1)}{c}$. If $c < \frac{d(d-1)}{c}$, $Pr[e(i, j)]$ is always smaller. Otherwise, only when $\frac{c}{\alpha} \leq \frac{d(d-1)}{c}$ can

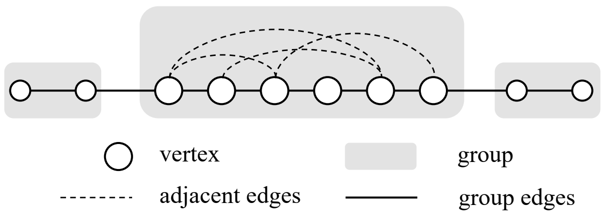


Fig. 3. Generating graphs with adjustable diameters

$\Pr[e(i,j)]$ become lower. Combining two conditions, we have:

$$d(d-1) \geq \frac{c^2}{\alpha}, \quad d \geq \frac{1+\sqrt{1+\frac{4c^2}{\alpha}}}{2}$$

For convenience, we can use $d \geq \sqrt{c^2/\alpha}$ to quickly check since $d \approx d-1$ when d is large. Therefore, if d is larger than this boundary, i.e., the vertices are too far away, the probability $\Pr[e(i,j)]$ is decreasing due to the influence of α. Correspondingly, other nearing vertices will have a higher probability. When the value of α is larger, the probability of each new edge being close to the edges generated in the previous round increases, which leads to a higher graph density. Though it is difficult to quantify the impact of α, our experiments show that increasing α ten-fold results in approximately twice as many edges being generated.

Table 4. Selected synthetic datasets

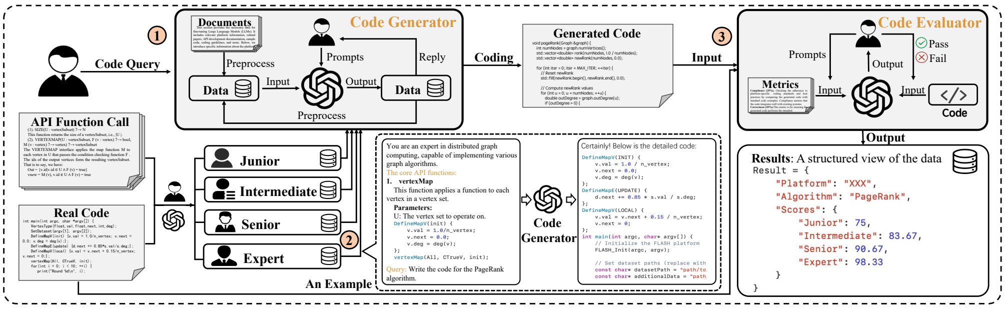
Datasets	**n**	**m**	**Density**	**Diameter**
S8-Std	3.6M	153M	2.4×10^{-5}	6
S8-Dense	1.2M	159M	2.2×10^{-4}	5
S8-Diam	3.6M	155M	2.4×10^{-5}	101
S9-Std	27.2M	1.42B	3.8×10^{-6}	6
S9-Dense	9.1M	1.47B	3.6×10^{-5}	5
S9-Diam	27.2M	1.48B	4.0×10^{-6}	102
S9.5-Std	77M	4.36B	1.5×10^{-6}	6
S10-Std	210M	12.62B	5.7×10^{-7}	6

4.2.2 Diameter Adjustment. The diameter is a crucial metric, particularly in the context of distributed algorithms, yet it is often overlooked in many synthetic datasets. Thus, we propose a method that helps adjust diameters to accommodate various needs.

Our approach restricts that most edges are within a group. As depicted in Figure 3, all vertices are organized into distinct groups. Then, edges are classified into two types: adjacent edges and group edges. Initially, adjacent edges are established between neighboring vertices to guarantee connectivity. Following this, group edges within each group are generated using FFT-DG. The diameter within each group remains relatively constant and independent of the group's size, averaging about 6, thus allowing control over the number of groups to produce a range of diameters:

$$group\_number = \frac{target\_diameter}{group\_diameter + 1}, group\_size = \frac{|V|}{group\_number}$$

The adjustment involves an additional check in Line 7 of Algorithm 1 that the iteration also breaks once the vertices with indices i and k do not belong to the same group as: $\left\lfloor \frac{i}{group\_size} \right\rfloor \neq \left\lfloor \frac{k}{group\_size} \right\rfloor$. Once k is too large to exceed the group, the generation process for u_i is finished.



```
void pageRank(Graph &graph) {
  int numNodes = graph.numVertices();
  std::vector<double> rank(numNodes, 1.0 / numNodes);
  std::vector<double> newRank(numNodes, 0.0);

  for (int iter = 0; iter < MAX_ITER; ++iter) {
    // Reset newRank
    std::fill(newRank.begin(), newRank.end(), 0.0);

    // Compute newRank values
    for (int u = 0; u < numNodes; ++u) {
      double outDegree = graph.outDegree(u);
      if (outDegree > 0) {
```

You are an expert in distributed graph computing, capable of implementing various graph algorithms.
The core API functions:
1. **vertexMap**
This function applies a function to each vertex in a vertex set.
Parameters:
U: The vertex set to operate on.

```
DefineMapV(init) {
    v.val = 1.0/n_vertex;
    v.next = 0.0;
    v.deg = deg(v);
};
vertexMap(All, CTrueV, init);
```

Query: Write the code for the PageRank algorithm.

Certainly! Below is the detailed code:

```
DefineMapV(INIT) {
    v.val = 1.0 / n_vertex;
    v.next = 0.0;
    v.deg = deg(v);
};
DefineMapE(UPDATE) {
    d.next += 0.85 * s.val / s.deg;
};
DefineMapV(LOCAL) {
    v.val = v.next + 0.15 / n_vertex;
    v.next = 0;
};
int main(int argc, char* argv[]) {
    // Initialize the FLASH platform
    FLASH_Init(argc, argv);

    // Set dataset paths (replace with
    const char* datasetPath = "path/to
    const char* additionalData = "path
```

Results: A structured view of the data

```
Result = {
    "Platform": "XXX",
    "Algorithm": "PageRank",
    "Scores": {
        "Junior": 75,
        "Intermediate": 83.67,
        "Senior": 90.67,
        "Expert": 98.33
    }
}
```

Fig. 4. LLM-based usability evaluation framework

4.3 Synthetic Datasets

Based on FFT-DG, we create eight synthetic datasets with differing scales, densities, and diameters, serving as the default synthetic datasets for our benchmark. Table 4 shows their details.

Specifically, we design four different scales, where the vertex numbers are correlated to the LDBC default setting, i.e., 3.6M, 27.2M, 77M, and 210M. We also provide two datasets with an enhanced density (i.e., 1.2M and 9.1M vertices and the same number of edges) and two datasets with a larger diameter (i.e., $\sim$ 100 with the same scale).

Each dataset is named based on its size scale and characteristic features. The size scale is quantified as the base-10 logarithm of the sum of the number of vertices and edges, denoted as $\log_{10}(|V| + |E|)$. The suffix *Std*, *Dense* and *Diam* represent standard social networks, dense networks and large-diameter networks, respectively.

5 Benchmarking API Usability

5.1 An Overlooked Benchmarking Scope

The API usability is a qualitative characteristic that evaluates how easy it is to learn and use [57, 68]. Highly usable APIs enhance productivity and reduce bugs, while initial interactions with poorly designed APIs can leave a lasting impression of complexity, potentially deterring users from embracing the platform. The most common evaluation method is empirical studies, where programmers are recruited to answer interview questions [19, 63]. However, this approach is costly and not scalable, as recruiting participants with adequate programming skills is challenging, and the number of possible usage scenarios for an API can be vast [19]. Consequently, most graph analytics platform benchmarks overlook API usability evaluations, instead reporting the lines of code required for specific tasks as an indicator of API usability [58, 85, 97]. Clearly, this approach is oversimplified and inadequate.

To address the shortcomings in existing API usability benchmarks, we propose a scalable, automated approach. Our idea is that since these shortcomings are primarily due to human factors, and LLMs have demonstrated the ability to match or surpass human abilities in many areas, why not use LLMs to replace humans in API usability evaluation? This approach eliminates the need for qualified programmers, cuts costs, and enables large-scale evaluation.

Our Solutions. To ensure LLMs familiarize each platform, we perform instruction-tuning using a comprehensive dataset, including platform-specific API documentation, research papers, and sample code. This enhances their ability to generate accurate code. However, determining the extent of the LLM's proficiency with different platforms is still challenging. To address this, we use multi-level prompts (from junior to expert) to simulate varying skill levels and stabilize the LLM's understanding. We also evaluate the generated code by comparing it with standard reference code and assessing its

Table 5. Performance evaluation metrics

Category	Metric	Description
Timing	**Upload Time**	Time required to read, convert, partition, and load graph data into memory.
	Running Time	Total time required to complete an algorithm execution task.
	Makespan	Overall time for graph operations, including reading, processing, and writing data.
Throughput	**Edges/sec**	Number of edges processed per second.
Scalability	**Speedup**	Rate of performance improvement with additional computational resources.
Robustness	**Stress Test**	Platform's stability and reliability under high-stress conditions.

correctness and readability, ensuring effective use of platform-specific APIs. Our proposed multi-level LLM-based usability evaluation framework is shown in Figure 4 and detailed in Section 5.2.

5.2 A Multi-Level LLM-Based Usability Evaluation Framework

Overview of the Framework. Our idea is to simulate programmers with varying skill levels by using LLMs with multi-level prompts. By analyzing the quality of the generated code, we can evaluate the usability of platform APIs. Specifically, we (1) perform instruction-tuning on LLMs to train the *Code Generator* for each platform by using their public documentation; (2) use the *Code Generator* to produce the code we requested by providing multi-level prompts; and (3) employ the instruction-tuned LLM-based *Code Evaluator* to score the generated code.

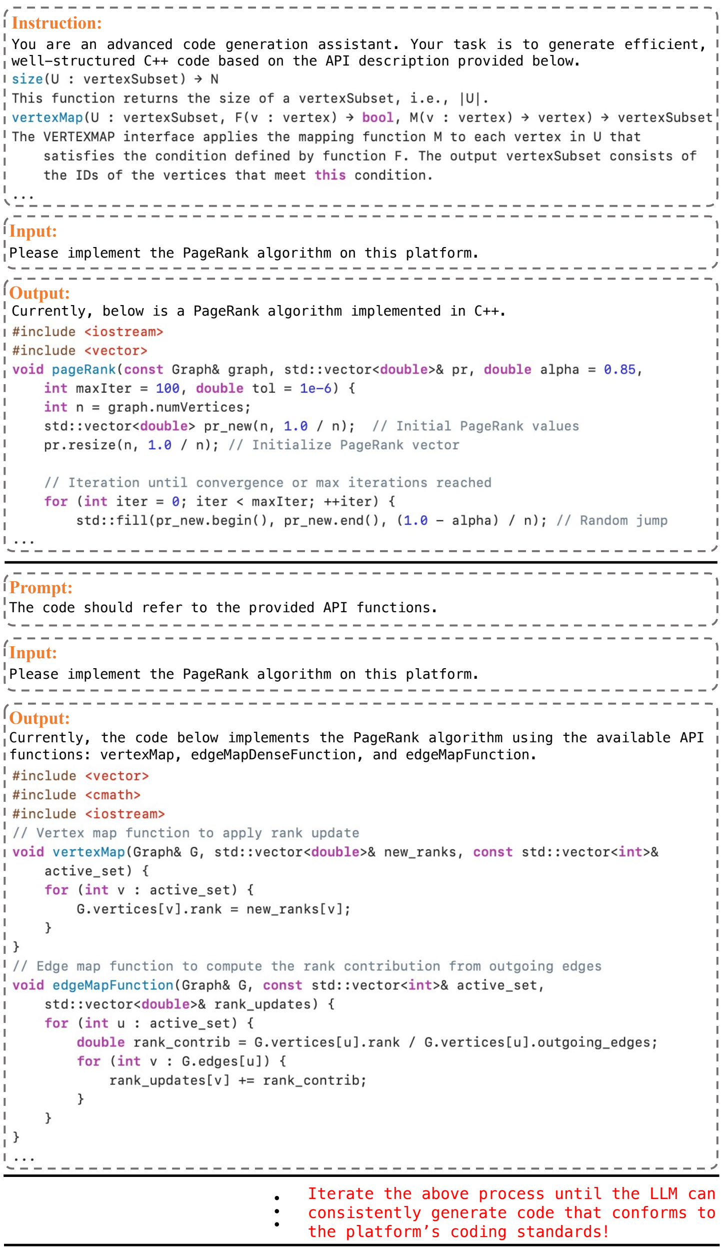
Step 1: Instruction-Tuning of LLMs. Step 1 aims to let LLMs generate code aligned with each platform's style, standards, and best practices, especially regarding platform-specific APIs by learning from publicly available documents. As shown in Figure 4 ①, we gather and preprocess data from each platform, including research papers, API documentation, sample code, and coding guidelines, enabling LLMs to learn specific coding styles and conventions. Human evaluators then review the generated results, providing feedback and prompts, which refine the data and the instruction-tuning process. This iterative process continues until the LLM consistently generates code that complies with platform standards.

Figure 5 illustrates how the LLM is trained to generate PageRank code on a specific platform. The example shows its code's evolution including from general C++ to platform-specific APIs. The LLM is given instructions and prompts to adjust its understanding of platform-specific functions, refining the generated code until it aligns with the platform's best practices, such as using functions like `vertexMap` and `edgeMapFunction`.

To ensure fairness and avoid bias during the code generation and evaluation process, all platform-specific identifiers in the API functions are anonymized. During the instruction-tuning process, platform-specific references, such as unique function names or parameters, are modified. This guarantees that LLMs evaluate the usability of APIs based on their general usability characteristics rather than familiarity with specific platforms.

Step 2: Multi-Level Prompts. Step 2 simulates programmers with varying expertise levels: junior, intermediate, senior, and expert, as shown in Figure 4 ②. The *Code Generator* uses varying prompts to generate corresponding code, and we evaluate difficulty by examining the generated code's quality, thereby assessing the platform's API usability. For example, high scores at lower-level prompts indicate ease of use and low learning costs of APIs on this platform. The prompt levels and their respective details are as follows:

- **Level 1 (Junior).** At this level, no specific technical details are provided. The prompt only consists of a description of the task, without any guidance on how to implement it.



Instruction:

```
You are an advanced code generation assistant. Your task is to generate efficient,
well-structured C++ code based on the API description provided below.
size(U : vertexSubset) → N
This function returns the size of a vertexSubset, i.e., |U|.
vertexMap(U : vertexSubset, F(v : vertex) → bool, M(v : vertex) → vertex) → vertexSubset
The VERTEXMAP interface applies the mapping function M to each vertex in U that
    satisfies the condition defined by function F. The output vertexSubset consists of
    the IDs of the vertices that meet this condition.
...
```

Input:

```
Please implement the PageRank algorithm on this platform.
```

Output:

```
Currently, below is a PageRank algorithm implemented in C++.
#include <iostream>
#include <vector>
void pageRank(const Graph& graph, std::vector<double>& pr, double alpha = 0.85,
    int maxIter = 100, double tol = 1e-6) {
    int n = graph.numVertices;
    std::vector<double> pr_new(n, 1.0 / n);  // Initial PageRank values
    pr.resize(n, 1.0 / n); // Initialize PageRank vector

    // Iteration until convergence or max iterations reached
    for (int iter = 0; iter < maxIter; ++iter) {
        std::fill(pr_new.begin(), pr_new.end(), (1.0 - alpha) / n); // Random jump
...
```

Prompt:

```
The code should refer to the provided API functions.
```

Input:

```
Please implement the PageRank algorithm on this platform.
```

Output:

```
Currently, the code below implements the PageRank algorithm using the available API
functions: vertexMap, edgeMapDenseFunction, and edgeMapFunction.
#include <vector>
#include <cmath>
#include <iostream>
// Vertex map function to apply rank update
void vertexMap(Graph& G, std::vector<double>& new_ranks, const std::vector<int>&
    active_set) {
    for (int v : active_set) {
        G.vertices[v].rank = new_ranks[v];
    }
}
// Edge map function to compute the rank contribution from outgoing edges
void edgeMapFunction(Graph& G, const std::vector<int>& active_set,
    std::vector<double>& rank_updates) {
    for (int u : active_set) {
        double rank_contrib = G.vertices[u].rank / G.vertices[u].outgoing_edges;
        for (int v : G.edges[u]) {
            rank_updates[v] += rank_contrib;
        }
    }
}
...
```

⋮ Iterate the above process until the LLM can consistently generate code that conforms to the platform's coding standards!

Fig. 5. An example of the instruction-tuning process for the code generator

- **Level 2 (Intermediate).** This level offers minimal technical information to guide the code generation. Basic prompts are given, including the names of core APIs and parameters.
- **Level 3 (Senior).** This level provides detailed usage instructions for the relevant APIs, including the names of the APIs and parameters, and a detailed introduction to them. Besides, some example code is provided to guide the usage of API functions.
- **Level 4 (Expert).** In addition to the detailed API instructions similar to the previous level, this level also includes the pseudo-code of the relevant algorithm.

To ensure fairness in our usability evaluation, we strictly rely on the most fundamental (i.e., lowest-level) APIs provided by each platform, such as `compute()` and `reducer()` in Pregel+ or `gather()`, `apply()`, and `scatter()` in PowerGraph. This approach ensures that the evaluation reflects the flexibility and extensibility of the API design itself rather than relying on simplified, high-level wrappers.

Step 3: Code Evaluation. Step 3 is crucial for determining the quality of the generated code and the API's usability. We first define the evaluation metrics. Correctness and readability are commonly used metrics in API usability evaluation [56, 62, 68]. Correct code indicates that the API is intuitive and that the provided documentation and examples effectively guide developers in implementing the desired functionality correctly. High readability suggests that the API's design promotes clear and concise coding practices. These two metrics are crucial for assessing usability. However, we observe that LLMs often exhibit "hallucination", focusing on general programming patterns or inventing nonexistent API functions while ignoring platform-specific APIs, typically due to unclear or ambiguous prompts. This limitation mirrors the behavior of human developers: less experienced programmers are more likely to make mistakes when dealing with poorly designed APIs. To address this, we introduce a new metric, compliance, which measures how closely the generated code aligns with standard code. This metric reflects the API's intuitiveness and accessibility for developers with varying skill levels. By assessing whether the API enables users to easily produce code that establishes best practices and standards, compliance provides an objective basis for scoring API usability. The details of evaluation metrics and their weight are as follows:

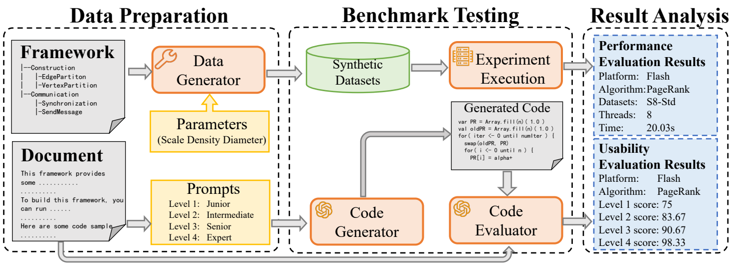
- **Compliance (35%).** Checking the adherence to platform-specific coding standards and best practices by comparing the generated code with standard code examples. Compliance ensures that the code integrates well with existing systems.
- **Correctness (35%).** This metric is for ensuring the generated code performs the intended task accurately. This includes verifying the logic of the code and the correctness of function calls.
- **Readability (30%).** This metric focuses on code clarity and maintainability. Readable code is easier to understand, modify, and debug. It should be well-structured, logically grouped, and follow consistent naming conventions.

A simple method would be to assign equal weights, but this leads to an indivisibility issue. Considering that compliance and correctness more directly reflect the LLM's understanding of platform APIs compared to readability, we assign slightly higher weights to compliance and correctness. Users can also customize the weight distribution based on their specific needs. During evaluation, we provide *Code Evaluator* with the evaluated and standard code and ask it to generate scores for each metric based on our provided guidelines. Figure 4③ illustrates the training process: we first provide detailed scoring criteria as LLM's knowledge base and set basic requirements instructions to get *Code Evaluator*. Then, we introduce some test codes and provide feedback based on the output results to optimize the Code Evaluator's instructions. We iterate this process until the LLM can produce stable and satisfactory evaluation results.

6 Overall Process of Our Benchmark

In this section, we provide a detailed overview of our benchmark's workflow, aiming to offer an in-depth evaluation of platforms from both performance and usability. As illustrated in Figure 6, the architecture of our benchmark includes three main stages: data preparation, benchmark testing and result analysis.

Data Preparation. In this stage, we generate and preprocess data for benchmarking. Using a flexible *Data Generator* (Section 4), we create synthetic graph datasets by varying parameters (e.g., scale, density, diameter) and convert them into platform-compatible formats. We also gather and



```
var PR = Array.fill(n)( 1.0 )
val oldPR = Array.fill(n)( 1.0 )
for( iter <- 0 until numIter ) {
  swap(oldPR, PR)
  for( i <- 0 until n ) {
    PR[i] = alpha+
```

Fig. 6. An overview of our benchmark

Table 6. Programming languages and computing models (Section 3.3) used in each platform

Platform	Language	Execution Model
GraphX	Scala	vertex-centric
PowerGraph	C++	edge-centric
Flash	C++	vertex-centric
Grape	C++ / Java	block-centric
Pregel+	C++	vertex-centric
Ligra	C++	vertex-centric
G-thinker	C++	subgraph-centric

preprocess platform resources—research papers, API documentation, and sample code—to meet the requirements of our usability evaluation framework (Section 5). Different prompts targeting platform-specific algorithms are prepared for the *Code Generator*, and to avoid bias, all platform-related information is concealed in both prompts and platform data.

Benchmark Testing. In this stage, we evaluate both performance and API usability. Using the *Experiment Executor*, we run selected core algorithms on synthetic datasets across various platforms. We measure metrics such as timing, throughput, scalability, and robustness (summarized in Table 5) to gauge each platform-algorithm combination's efficiency and effectiveness.

In the API usability evaluation stage, the *Code Generator* uses predefined prompts to produce code, which is combined with the platform-provided code and scored by the *Code Evaluator*. To reduce bias, we repeat this process multiple times and average the results to obtain the final score.

Result Analysis. In this stage, we aggregate and analyze both performance outcomes and API usability scores to reveal each platform's strengths and weaknesses. An example of the experimental results analysis is provided in Section 9.

7 Experimental Setup

7.1 Environment

We evaluate seven widely used graph analytics platforms. Each platform employs distinct programming languages, storage formats, communication methods, and computation models, which is summarized in Table 6.

- **GraphX (GX)** [23] is a component in Spark adapted for graph-parallel computation, which provides vertex-centric computing model using Spark Resilient Distributed Datasets (RDD) APIs.
- **PowerGraph (PG)** [43] is an edge-centric distributed graph analysis platform, addressing the difficulties including load imbalance in graph algorithms over real-world power-law graphs.
- **Flash (FL)** [41] is a distributed platform, supporting various complex graph algorithms including clustering, centrality, traversal, etc. It extends the vertex-centric model with global vertex status to make it easy to program complex graph algorithms.

Table 7. Experimental methodology

Aspects	Section	Algorithms	Datasets	#threads	#machines
Algorithm Impact	Section 8.2	All	S8-Std, S8-Dense, S8-Diam	32	1
Statistics Impact					
Scalability Sensitivity	Section 8.3	PR, SSSP, TC	S8-Std, S8-Dense, S8-Diam	1, 2, 4, 8, 16, 32	1
			S9-Std, S9-Dense, S9-Diam	32	1, 2, 4, 8, 16
Throughput	Appendix [1]	PR, SSSP, TC	S8-Std, S8-Dense, S8-Diam, S9-Std, S9-Dense, S9-Diam	32	16
Stress Test	Appendix [1]	PR	S8-Std, S9-Std, S9.5-Std, S10-Std	32	16
Usability Evaluation	Section 8.4	All	–	–	–

- Grape **(GR)** [16] is a block-centric parallel platform, which is designed to execute existing (textbook) sequential graph algorithms with only minor modifications in the distribution context.
- **Pregel+ (PP)** [92] extends the original vertex-centric platform Pregel [46] by introducing vertex mirroring and efficient message reduction, making it effective for power-law and dense graphs.
- **Ligra (LI)** [73] is a lightweight vertex-centric processing platform for shared memory on a single machine with multiple cores. Ligra is applicable if one machine can allocate the whole graph.
- **G-thinker (GT)** [94] is a subgraph-centric platform specialized for graph mining problems, which can create and schedule subgraphs with arbitrary shapes and sizes.

All experiments are evaluated on a cluster with 16 machines. Each machine has 4 Intel® Xeon® Platinum 8163 @ 2.50GHz CPUs, 512 GB memory, and 3 TB disk space. The 16 machines are connected with a 15 Gbps LAN network.

7.2 Algorithms

We utilize all eight algorithms outlined in Section 3 to evaluate the platforms, with the following settings: For PageRank (PR) and Label Propagation Algorithm (LPA), the maximum number of iterations is set to 10. For Triangle Counting (TC) and k-Clique (KC), only the counting results are reported. In Single Source Shortest Path (SSSP) and Betweenness Centrality (BC), the source vertex is set to 0. For Connected Components (WCC), we use undirected networks (i.e., weakly connected) to identify all connected components. In CD, the minimum coreness value starts at 1 and is gradually increased until all vertices are removed. Detailed implementation descriptions for all algorithms can be found in the artifact.

7.3 Datasets

We test seven synthetic datasets, as detailed in Section 4.3. Note that evaluations on real-life datasets are similar to the LDBC benchmark [28] and are therefore not the focus of this work. As mentioned above, these datasets are in four scales: S8, S9, S9.5, and S10, with S10 being the largest one. Additionally, there are three variants for S8 and S9: Std, Dense, and Diam. Here, Std represents the standard dataset (α=10), Dense denotes a denser dataset (α=1000), and Diam indicates a dataset with a larger diameter (α=10).

7.4 Methodology

Our evaluation methodology, detailed in Table 7, encompasses six aspects. Unless stated otherwise, the experiments are conducted using three algorithms: PR, SSSP, and TC. These algorithms represent the Iterative, Sequential, and Subgraph, respectively, as discussed in Section 3.3.

- **Algorithm Impact** evaluates a platform's ability to support different graph algorithms. We use 32 threads on a single machine to run all algorithms on three S8 datasets with varying statistics.
- **Statistics Impact** assesses platform sensitivity to various dataset characteristics by running all algorithms on the three S8 datasets. This experiment highlights platform optimizations that

Table 8. Jensen-Shannon Divergence between LiveJournal and FFT-DG/ LDBC-DG's datasets

Generator	CC	TPR	BR	Diam	Cond	Size
FFT-DG	0.108	0.057	0.057	0.053	0.054	0.069
LDBC-DG	0.201	0.097	0.087	0.236	0.183	0.133

enable effective handling datasets with diverse characteristics, which may not be evident when using a single dataset alone.

- **Scalability Sensitivity** assesses both scale-up and scale-out performance. For scale-up, we increase the number of threads on a single machine (1, 2, 4, 8, 16, 32). For scale-out, we increase the number of machines (1, 2, 4, 8, 16), each utilizing 32 threads. To evaluate the platform's sensitivity to different datasets, we use datasets with varying statistics, including the larger S9 datasets while testing scale-out performance.
- **Throughput** measures data processing capacity. We run the default algorithms on $S8$ and $S9$ in 16 machines, each configured with 32 threads.
- **Stress Test** aims to determine the largest dataset that each platform can handle on 16 machines. To achieve this, we run the PageRank (PR) algorithm on datasets S8-Std, S9-Std, S9.5-Std, and S10-Std until the platform is unable to process the dataset, revealing its limitations.
- **Usability Evaluation** involves the pioneering subjective assessment of the platform's API usability using our LLM-based evaluation framework, as described in Section 5.2. This evaluation is powered by the GPT-4o model [3]. To ensure the evaluation is convincing, we limit our evaluation to code provided by the platform and the original authors, excluding any code we have written ourselves.

The TPC-H price/performance is also an important metric. However, in our experiments, all systems are deployed on the same cluster with 16 machines, ensuring the cost remains constant. Moreover, the processing time and throughput evaluated in the paper partially reflect the system's performance. Thus, the throughput indirectly reflects the price/performance metric, where a higher throughput indicates a better price/performance ratio.

8 Experimental Result

8.1 Generation Similarity and Efficiency

Generation Similarity. To illustrate the better alignment with real-world graph characteristics, we follow the previous evaluation methodology [64], i.e., generate communities over the social network and present the distribution of community statistics in the dataset. We choose the LiveJournal [38] dataset as the ground truth and select six statistics, Clustering Coefficient, Triangle Participation Ratio, Bridge Ratio, Diameter, Conductance, and Size. We use the Jensen-Shannon Divergence to evaluate the similarity and the result in Table 8 suggests that our datasets achieve 2x lower divergence on average, verifying that FFT-DG can generate social networks that follow the real-world distribution and are similar to real-world datasets. Figure 7 presents the distribution of three major statistics with complete results provided in the Appendix [1].

We also generate FFT-DG and LDBC-DG datasets with the same size as the LiveJournal and evaluate the performance of PR and SSSP on all six supported platforms. For FFT-DG, we tune the density factor α, and for LDBC-DG, we reduce the degree of all vertices. The performance and relative difference are shown in Figure 8 and Table 9, respectively. Compared to LDBC-DG, the FFT-DG dataset shows equal (or slightly better) runtime similarity to the real-world dataset (LiveJournal) on most platforms, with a difference within 25% (except for Ligra). In fact, the total workload on distributed platforms is quite unpredictable so this evaluation can validate the performance similarity.

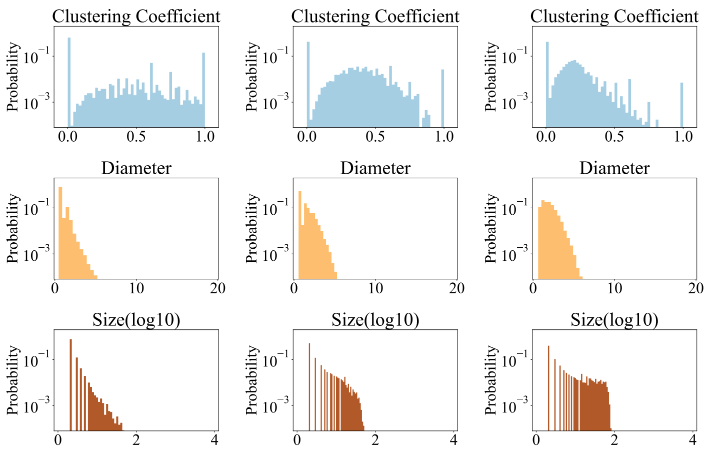


Fig. 7. Distribution of dataset communities statistics

Table 9. Relative Difference between LiveJournal and FFT-DG/ LDBC-DG's datasets

Algo.	Generator	GX	PG	FL	GR	PP	LI
PR	FFT-DG	12%	25%	15%	12%	1%	48%
	LDBC-DG	12%	11%	33%	38%	11%	57%
SSSP	FFT-DG	11%	25%	15%	11%	10%	57%
	LDBC-DG	37%	28%	43%	24%	24%	56%

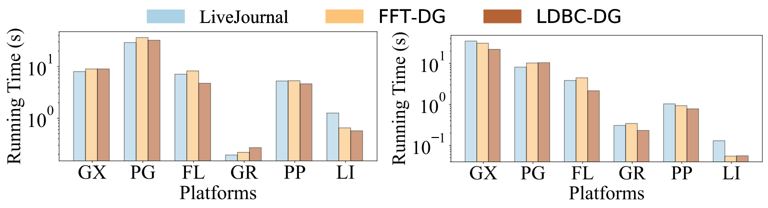


Fig. 8. Running time of PR and SSSP on three datasets

Generation Efficiency. The efficiency evaluation involves generating multiple datasets with the density factor α varying in $\{1, 10, 100, 1000\}$. Figure 9 illustrates the total number of trials and generated edges per second, respectively. FFT-DG demonstrates consistent throughput as density increases, requiring about 1.5 trials per edge, while LDBC-DG necessitates more than 8 trials per edge. Although LDBC-DG exhibits a faster sampling process, capable of executing up to 2 times the number of trials per second, its edge generating speed is still about 2.2 times slower than FFT-DG.

8.2 Algorithm & Statistics Impact

We present experimental results of algorithms and statistics impact in Figure 10 and analyze them collectively.

Algorithm Coverage across Platforms. All experiment codes are sourced from official documentation, third-party repositories, or implemented by us to the best of our ability. Out of the total 8 (algorithms) $\times$ 7 (platforms) = 56 cases, we successfully evaluate 49 cases. An algorithm's suitability for a given computing model (Section 3.3) affects whether it is feasible and/or efficient to implement

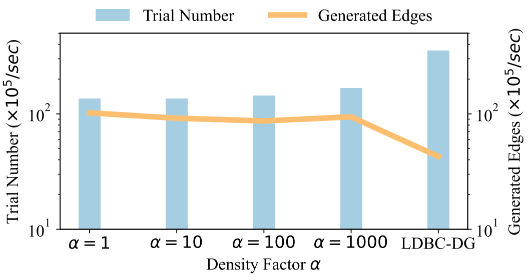


Fig. 9. Generated edge number and generating time with different density factor α.

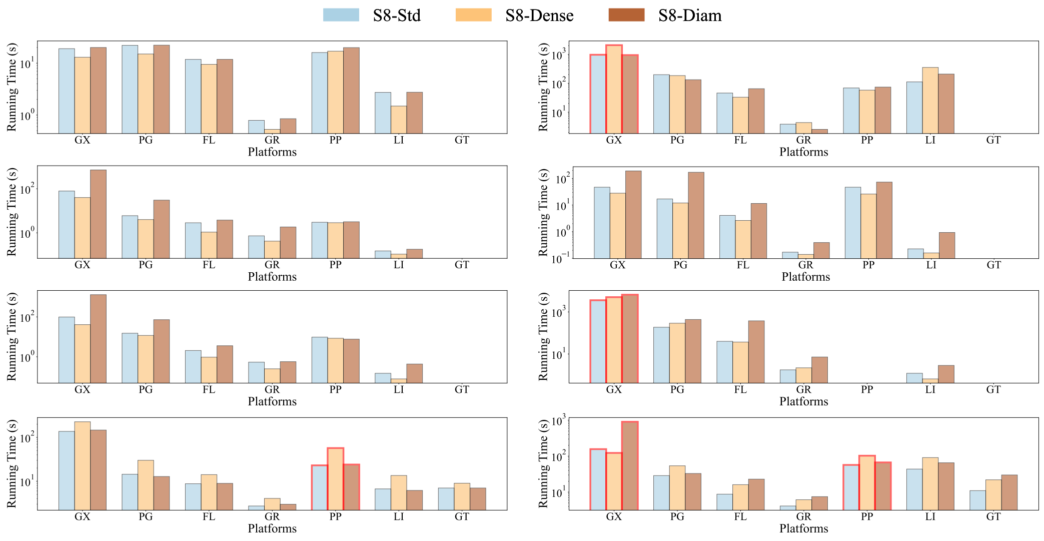


Fig. 10. Running time of eight algorithms on datasets S8-Std, S8-Dense, S8-Diam

the algorithm on different platforms. Among the 7 unimplemented cases: Pregel+'s interface lacks support for managing variables like coreness values required by CD across supersteps; PR, LPA, SSSP, WCC, BC, and CD cannot be implemented on G-thinker, a subgraph-centric platform that does not support the iterative control flow needed by these algorithms. Additionally, GraphX struggles with LPA, CD, and KC due to the overhead for adapting the Spark RDD APIs; Pregel+ lacks push/pull optimizations found in newer platforms like Flash. Consequently, it faces high computation overhead with subgraph algorithms (TC and KC). To manage these inefficiencies, we use 16 machines instead of one single machine for these cases (indicated by **red bars** in Figure 10).

Generally, eight algorithms exhibit three computation patterns:

Iterative Algorithms (PR **and** LPA) perform similarly on S8-Std and S8-Diam, suggesting that these algorithms are not sensitive to diameter. However, there are noticeable differences in performance on S8-Dense. For PR, all platforms run faster on S8-Dense as expected, as PR distributes its workload across vertices, and the denser dataset has fewer vertices with the same number of edges. The performance of LPA is affected by implementation, which requires hash tables to store vertex labels. Specifically, GraphX experiences significant performance degradation when merging hash tables from different vertices. In contrast, other platforms maintain a local hash table and can merge received hash tables to the local one, avoiding much redundant computation.

Sequential Algorithms (SSSP, WCC, BC, CD**)** generally perform better on S8-Dense but worse on S8-Diam, indicating sensitivity to both density and diameter. Similar to PR, workloads are distributed over vertices, resulting in improved performance on S8-Dense. However, their sequential nature and the increased synchronization requirements in S8-Diam result in reduced performance.

For SSSP and BC, both vertex-centric and edge-centric platforms like GraphX and PowerGraph cost too much time on multiple synchronizations, while the block-centric platform like Grape is not sensitive to the diameter. Since only a portion of vertices are active in each iteration, vertex-centric optimizations like the push-pull model and the vertex subset technique in Flash and Ligra can also improve efficiency. Besides, Pregel+ performs equally well on three datasets, showing a strong workload balance.

Flash and Pregel+ offer standard Pregel-like APIs that support global communication, i.e., send messages to any vertices, enabling some advanced algorithms such as HashMin [67] and Shiloach-Vishkin [65, 80, 93] for WCC and reducing iteration rounds significantly. Block-centric models like Grape can even directly call sequential implementation, e.g., Disjoint Set, for dynamic WCC. In contrast, some vertex-centric platforms (e.g., GraphX) and the edge-centric model can only send messages to neighbors.

Unlike other sequential algorithms, the performance of CD exhibits varying trends across different platforms when executed on denser datasets. Since the coreness value could be very large (especially in S8-Dense) and multiple iterations may be required for each coreness value, GraphX is extremely slow even we use 16 machines. Additionally, for each coreness value, platforms like GraphX and PowerGraph need to activate all vertices, while Flash and Ligra can maintain the activated vertices, resulting in better performance.

Subgraph Algorithms (TC, KC**)** require substantial communication to broadcast neighbor lists and massive computation to count subgraphs, where block-centric (Grape) and subgraph-centric (G-thinker) models can perfectly reduce overhead. For TC, all platforms count the common neighbors of two vertices of each edge, where denser datasets have larger neighbor lists, resulting in poorer performance on S8-Dense. Unlike TC, the performance of KC on S8-Diam is also poor. The reason is that to generate a long-diameter graph with the same scale, the local partition of the graph must be relatively dense, leading to substantially more cliques. Therefore, the bottleneck is communication for PowerGraph, Pregel+, Ligra, but computation for GraphX, Flash, Grape, G-thinker.

8.3 Scalability Sensitivity

Varying Number of Threads. We first examine the scale-up performance on a single machine with increasing threads: 1, 2, 4, 8, 16, and 32. GraphX requires at least four threads for PR and two threads for SSSP to operate effectively, so we only exclude TC.

Figure 11 illustrates the scale-up performance of PR, SSSP, and TC across platforms. All platforms exhibit a clear decreasing trend in execution time as more threads are used. The detailed scaling factors, calculated as the ratio of best performance to single-thread performance, are presented in Table 10. Grape achieves the best scale-up performance, with a scaling factor of up to 37×, followed by Ligra, G-thinker and Pregel+, which have an average scaling factor of 24.9×, 23.3× and 19.6× . The scaling factors for Flash, GraphX and PowerGraph range between 1× and 12×.

The scale-up performance varies significantly across the three algorithms. The scaling factor for TC is higher than PR and SSSP. This confirms our previous categorization: TC requires no synchronization, allowing all computations to be parallelized, while PR consists of k iterations and k synchronizations. Sequential algorithms like SSSP require multiple synchronizations, making its scalability the worst among the three algorithms. Besides, the scale-up performance is sensitive to dataset characteristics. Most platforms scale better on S8-Dense but perform worse on S8-Diam than on S8-Std.

Table 10. Speed up (threads)

Algo.	Dataset	GX	PG	FL	GR	PP	LI	GT
PR	S8-Std	3.8	5.1	8.2	25.3	29.3	32.2	—
	S8-Dense	3.8	7.8	4.5	25.2	22.6	34.9	—
	S8-Diam	3.6	2.2	8.2	24.2	18.9	32.0	—
SSSP	S8-Std	6.9	5.0	9.3	23.5	14.7	17.8	—
	S8-Dense	7.8	5.8	8.5	19.7	16.4	18.2	—
	S8-Diam	6.7	0.9	10.2	13.2	15.8	22.4	—
TC	S8-Std	—	10.7	18.7	37.2	—	21.3	19.7
	S8-Dense	—	10.5	22.2	27.5	—	22.4	30.1
	S8-Diam	—	9.4	18.1	29.6	—	23.5	20.0

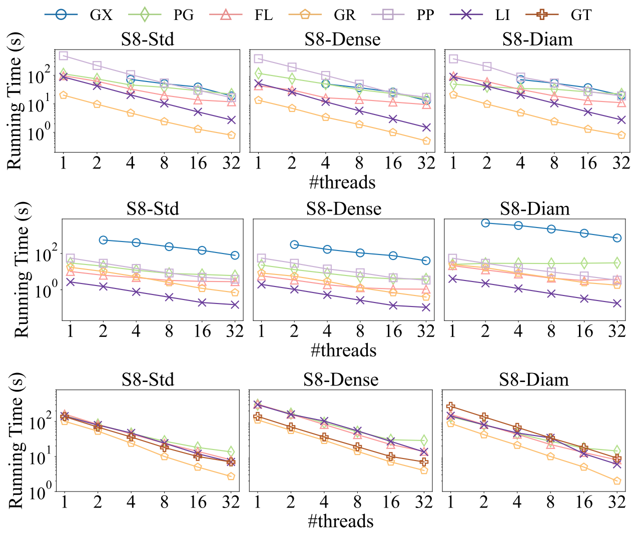


Fig. 11. Running time varying #threads (Scale = 8)

Table 11. Speed up (machines)

Algo.	Dataset	GX	PG	FL	GR	PP	GT
PR	S9-Std	3.2	2.3	0.8	5.8	5.2	—
	S9-Dense	3.8	2.2	1.0	11.5	6.1	—
	S9-Diam	3.0	2.4	0.8	6.1	5.4	—
SSSP	S9-Std	1.8	2.6	1.2	1.7	3.2	—
	S9-Dense	2.2	2.9	1.3	3.3	5.0	—
	S9-Diam	—	1.4	2.0	0.5	3.9	—
TC	S9-Std	—	—	3.3	3.2	—	3.1
	S9-Dense	—	—	2.6	4.1	—	2.7
	S9-Diam	—	—	4.7	3.9	—	3.1

Varying Number of Machines. We evaluate scale-out performance by increasing the number of machines to 2, 4, 8, and 16, using larger S9 datasets. GraphX fails to compute SSSP on S9-Diam, and GraphX and PowerGraph encounter Out-Of-Memory errors on TC. Ligra is not tested as it runs only on a single machine. Figure 11 shows the performance, and scaling factors, calculated as the ratio of best to single-machine performance, are summarized in Table 11.

All platforms show worse scale-out performance than scale-up, likely due to the high overhead of inter-machine communication. Pregel+ performs significantly better in scale-out than the others.

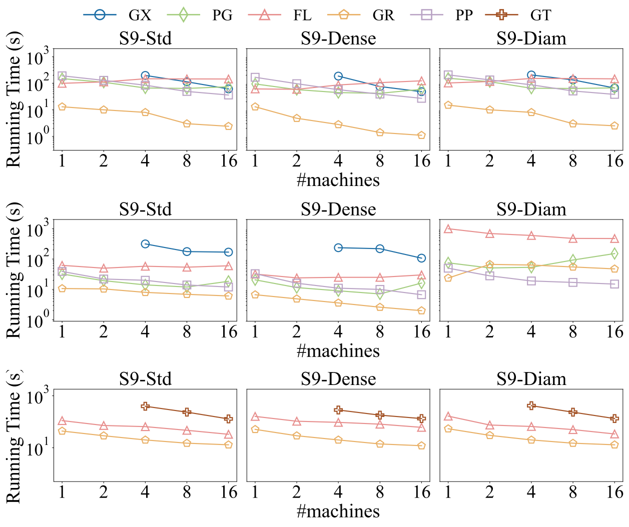


Fig. 12. Running time varying #machines (Scale = 9)

Notably, Grape performs well on a single machine, but its performance gains saturate when scaling to multiple machines due to communication overhead. The scalability trends, influenced by algorithms and dataset characteristics, are similar to those in the scale-up experiments and won't be further discussed.

8.4 Usability Evaluation

We use our usability evaluation framework to simulate programmers with varying expertise levels and evaluate the usability of APIs by analyzing the generated code. Figure 13 shows the usability scores for various platforms with different prompt levels (from junior to expert). To validate our framework's effectiveness, we invited over 80 students and developers with varying levels of familiarity with graph algorithms to assess the quality of code generated by our *Code Generator* at the "Intermediate" and "Senior" prompt levels. Platform-related information was masked for the human reviewers to ensure unbiased evaluations. Table 12 compares our framework's results with the human reviews.

From Figure 13, we see that GraphX achieves the highest scores across all expertise levels, with particularly outstanding scores from senior and expert users, indicating that its API design is highly user-friendly, providing an excellent experience for both novice and advanced users. PowerGraph and Pregel+ show balanced and relatively high usability scores, especially among junior and intermediate users, suggesting their APIs are intuitive for beginners. In contrast, Grape has the lowest scores from junior users, reflecting a steep learning curve and a less beginner-friendly design. However, its scores rise significantly for senior and expert users, indicating it offers robust functionality once users gain experience. Finally, Flash, Ligra, and G-thinker share similar patterns—lower scores at the junior level and higher scores at the senior and expert levels. While their abstraction of graph traversal interfaces can simplify programming, it may cause comprehension difficulties for beginners.

The results from human reviewers (Table 12) show a similar relative ranking to those of our framework. GraphX consistently receives the highest ranking, while Grape is ranked the lowest, with other platforms ranking in between.

Discussions of Scoring Difference. Human evaluation relies on a five-point scale (1–5) to keep the process manageable for human reviewers. In contrast, our LLM-based framework applies more

Table 12. Comparison of code evaluation results: LLM vs. human evaluations from over 80 students and developers (numbers in parentheses represent rankings)

Prompt	Eval.	GX	PG	FL	GR	PP	LI	GT
Intermediate	LLM	81.0(1)	77.0(2)	70.3(5)	68.5(7)	73.3(3)	72.7(4)	70.0(6)
	Human	77.4(1)	62.8(5)	68.8(3)	57.2(7)	70.3(2)	67.6(4)	61.7(6)
Senior	LLM	91.0(1)	80.6(6)	80.8(5)	77.5(7)	84.2(2)	82.1(3)	82.0(4)
	Human	78.2(1)	61.6(5)	74.6(2)	56.8(7)	72.0(3)	72.0(4)	65.7(6)

detailed evaluation rules with multiple dimensions, leading to finer-grained evaluation. As a result, the absolute scores produced by the LLM-based framework may differ from human evaluations. However, our focus is on the relative rankings and trends between LLM and human evaluation, rather than absolute score values. To assess the similarity of rankings between the two methods, we use Spearman's Rho, a statistical measure that quantifies the degree of correlation between two sets of ranked data, which ranges from -1 to 1, with values closer to 1 indicating stronger consistency between the rankings. The results, 0.75 for Intermediate and 0.714 for Senior, indicate that our framework's ranking closely mirrors human judgment. We observe cases where human rankings diverge from the LLM's, such as in the evaluation of PowerGraph at the "Intermediate" level. We speculated that the LLM may favor PowerGraph 's code due to its well-formulated APIs in this case. This presents an interesting future work to further fine-tune the LLM evaluator.

8.5 Summary

The experimental evaluation verifies the effectiveness of our benchmark, indicating the performance of a platform is affected by the different algorithms and datasets. For instance, Grape and G-thinker perform well in handling subgraph algorithms, while Grape and Ligra excel in sequential algorithms. Pregel+ shows balanced performance across different datasets, while GraphX, Flash, and PowerGraph are more sensitive. Additionally, our usability evaluation examines platform API usability, revealing their friendliness to developers in daily use. Based on the trade-off between performance and usability, we offer valuable insights for users in platform selection, which will be presented in Section 9.

9 Platform Selection Guide

Based on the results in Section 8, Figure 14 shows the performance of all platforms across different metrics. According to the area covered by each platform, the ranking is as follows: Pregel+ > Grape > GraphX > G-thinker > Flash > PowerGraph > Ligra. Pregel+ stands out as the top-performing platform overall, performing well on all metrics without any notable weaknesses and ranking in the top three for nearly all categories. Grape excels in throughput, machine speed-up, thread speed-up, and stress tests, showing strong performance and scalability. However, it scores the lowest in the three usability evaluation metrics, indicating a steep learning curve and complex API. In contrast, GraphX achieves the highest scores in compliance, correctness, and readability, highlighting its exceptional API usability. However, it performs poorly in thread speed-up, throughput, and stress test, indicating limited performance and scalability. G-thinker delivers a balanced performance in both usability and efficiency but supports fewer algorithms, leading to a lower algorithm coverage score and a mid-tier ranking. Flash and PowerGraph also shows balanced performance, but their individual rankings for each metric are not particularly outstanding, resulting in lower overall rankings. Ligra performs well for single-machine tasks, especially in thread speed-up, but ranks lowest overall due to its lack of support for distributed computing.

GraphX is ideal for users of all levels, as long as performance and scalability aren't top priorities, thanks to its excellent usability. Pregel+, Flash and PowerGraph are also suitable for most users,

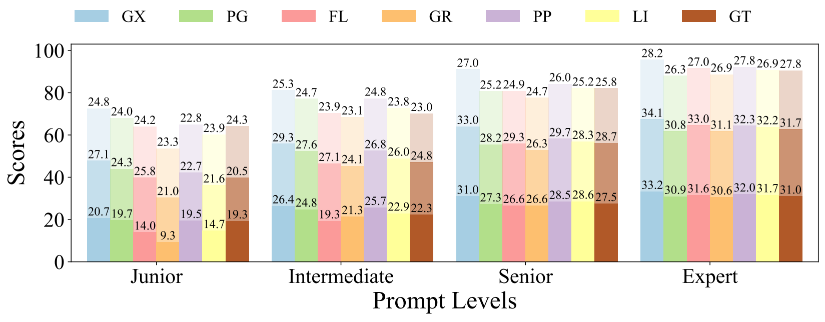


Fig. 13. Usability scores of varying platforms. The colors from deeper to lighter shades represent the scores for Compliance, Correctness, and Readability, respectively.

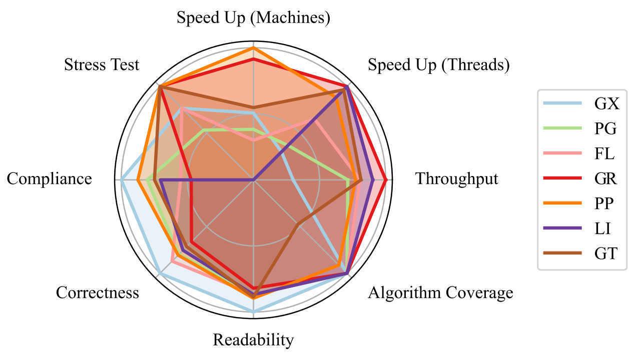


Fig. 14. Comprehensive comparison across platforms

especially beginners and intermediates, due to their balanced usability and good performance with large data. For those needing strong performance, G-thinker and Ligra provide exceptional performance. However, G-thinker supports fewer algorithms and is best suited for compute-intensive tasks, while Ligra provides robust interfaces and extensive algorithm support, excelling in single-machine tasks. Due to its lack of support for distributed computing, Ligra is unsuitable for large-scale or multi-machine workloads. For maximum performance and scalability, Grape is recommended despite its steeper learning curve, as it delivers significant efficiency once mastered.

10 Conclusion

In this paper, we present a new benchmark for large-scale graph analytics platforms, featuring eight algorithms and the FFT-DG, which improves dataset generation by adjusting scale, density, and diameter. We also introduce a multi-level LLM-based framework for evaluating API usability, marking the first use of such metrics in graph analytics benchmarks. Our extensive experiments assess both performance and usability, offering valuable insights for developers, researchers, and practitioners in platform selection.

Acknowledgments

We thank the 80 anonymous human experts for their valuable contributions to the usability evaluation. Long Yuan is supported by National Key R&D Program of China 2022YFF0712100, NSFC62472225. Peng Cheng is supported by NSFC 62102149. Xuemin Lin is supported by NSFC U2241211 and U20B2046.

References

[1] [n. d.]. Graph-Analytics-Benchmarks. https://anonymous.4open.science/r/Graph-Analytics-Benchmarks-C144

[2] [n. d.]. Neo4j-APOC. https://github.com/neo4j/apoc

[3] 2024. GPT-4o. https://chatgpt.com/?model=gpt-4o

[4] Khaled Ammar and M. Tamer Özsu. 2013. WGB: Towards a Universal Graph Benchmark. In *Advancing Big Data Benchmarks - Proceedings of the 2013 Workshop Series on Big Data Benchmarking, WBDB.cn, Xi'an, China, July 16-17, 2013 and WBDB.us, San José, CA, USA, October 9-10, 2013 Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8585)*, Tilmann Rabl, Hans-Arno Jacobsen, Raghunath Nambiar, Meikel Poess, Milind A. Bhandarkar, and Chaitanya K. Baru (Eds.). Springer, 58–72.

[5] Mehdi Azaouzi and Lotfi Ben Romdhane. 2017. An evidential influence-based label propagation algorithm for distributed community detection in social networks. *Procedia computer science* 112 (2017), 407–416.

[6] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *science* 286, 5439 (1999), 509–512.

[7] Nina Berry, Teresa Ko, Tim Moy, Julienne Smrcka, Jessica Turnley, and Ben Wu. 2004. Emergent clique formation in terrorist recruitment. In *The AAAI-04 Workshop on Agent Organizations: Theory and Practice*. 1198–1208.

[8] Luciana S Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. 2006. Counting triangles in data streams. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 253–262.

[9] Massimiliano Caramia and Giovanni Felici. 2006. Mining relevant information on the Web: a clique-based approach. *International Journal of Production Research* 44, 14 (2006), 2771–2787.

[10] Giuliana Carullo, Aniello Castiglione, Alfredo De Santis, and Francesco Palmieri. 2015. A triadic closure and homophily-based recommendation system for online social networks. *World Wide Web* 18 (2015), 1579–1601.

[11] Vinicius Dias, Carlos HC Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*. 1357–1374.

[12] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959), 269–271.

[13] Paul Erdos, Alfréd Rényi, et al. 1960. On the evolution of random graphs. *Publ. math. inst. hung. acad. sci* 5, 1 (1960), 17–60.

[14] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[15] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. 2017. GRAPE: Parallelizing sequential graph computations. In *43rd International Conference on Very Large Data Bases*. Very Large Data Base Endowment Inc., 1889–1892.

[16] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. 2017. Parallelizing Sequential Graph Computations. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, 495–510.

[17] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing sequential graph computations. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 1–39.

[18] Umer Farooq, León Welicki, and Dieter Zirkler. 2010. API usability peer reviews: a method for evaluating the usability of application programming interfaces. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems, CHI 2010, Atlanta, Georgia, USA, April 10-15, 2010*, Elizabeth D. Mynatt, Don Schoner, Geraldine Fitzpatrick, Scott E. Hudson, W. Keith Edwards, and Tom Rodden (Eds.). ACM, 2327–2336.

[19] Umer Farooq and Dieter Zirkler. 2010. API peer reviews: a method for evaluating usability of application programming interfaces. In *Proceedings of the ACM conference on Computer supported cooperative work*. 207–210.

[20] Sebastian Forster and Danupon Nanongkai. 2018. A faster distributed single-source shortest paths algorithm. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 686–697.

[21] Kuang Gao, Guocai Yuan, Yang Yang, Ying Fan, and Wenbin Hu. 2022. MEBC: social network immunization via motif-based edge-betweenness centrality. *Knowledge and Information Systems* 64, 5 (2022), 1263–1281.

[22] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 17–30.

[23] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 599–613.

[24] Graph500 Executive Committee. 2010. Graph 500 Benchmark. https://graph500.org/.

[25] Thomas Grill, Ondrej Polácek, and Manfred Tscheligi. 2012. Methods towards API Usability: A Structural Analysis of Usability Problem Categories. In *Human-Centered Software Engineering - 4th International Conference, HCSE 2012, Toulouse, France, October 29-31, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7623)*, Marco Winckler, Peter Forbrig, and Regina Bernhaupt (Eds.). Springer, 164–180.

[26] Aric Hagberg, Pieter J Swart, and Daniel A Schult. 2008. *Exploring network structure, dynamics, and function using NetworkX.* Technical Report. Los Alamos National Laboratory (LANL), Los Alamos, NM (United States).

[27] Paul W Holland and Samuel Leinhardt. 1971. Transitivity in structural models of small groups. *Comparative group studies* 2, 2 (1971), 107–124.

[28] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J. Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter A. Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (2016), 1317–1328.

[29] Gábor Iván and Vince Grolmusz. 2011. When the Web meets the cell: using personalized PageRank for analyzing protein interaction networks. *Bioinformatics* 27, 3 (2011), 405–407.

[30] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems.* 1–16.

[31] Zhaoyan Jin, Dianxi Shi, Quanyuan Wu, Huining Yan, and Hua Fan. 2012. Lbsnrank: personalized pagerank on location-based social networks. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing.* 980–987.

[32] Aisan Kazerani and Stephan Winter. 2009. Can betweenness centrality explain traffic flow. In *12th AGILE international conference on geographic information science.* Germany: Leibniz Universität Hannover, 1–9.

[33] Nicolas Kourtellis, Tharaka Alahakoon, Ramanuja Simha, Adriana Iamnitchi, and Rahul Tripathi. 2013. Identifying high betweenness centrality nodes in large social networks. *Social Network Analysis and Mining* 3 (2013), 899–914.

[34] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 31–46.

[35] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. 2008. Benchmark graphs for testing community detection algorithms. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 78, 4 (2008), 046110.

[36] Chin Yang Lee. 1961. An algorithm for path connections and its applications. *IRE transactions on electronic computers* 3 (1961), 346–365.

[37] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker graphs: an approach to modeling networks. *Journal of Machine Learning Research* 11, 2 (2010).

[38] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[39] Qingquan LI, Zhe ZENG, Bisheng YANG, et al. 2010. Betweenness centrality analysis for urban road networks. *Geomatics and Information Science of Wuhan University* 35, 1 (2010), 37–41.

[40] Xue Li, Ke Meng, Lu Qin, Longbin Lai, Wenyuan Yu, Zhengping Qian, Xuemin Lin, and Jingren Zhou. 2023. Flash: A framework for programming distributed graph processing algorithms. In *2023 IEEE 39th International Conference on Data Engineering (ICDE).* IEEE, 232–244.

[41] Xue Li, Ke Meng, Lu Qin, Longbin Lai, Wenyuan Yu, Zhengping Qian, Xuemin Lin, and Jingren Zhou. 2023. Flash: A Framework for Programming Distributed Graph Processing Algorithms. In *2023 IEEE 39th International Conference on Data Engineering (ICDE).* 232–244.

[42] Zhao Li, Xia Chen, Junshuai Song, and Jun Gao. 2022. Adaptive label propagation for group anomaly detection in large-scale networks. *IEEE Transactions on Knowledge and Data Engineering* 35, 12 (2022), 12053–12067.

[43] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence (UAI).*

[44] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment* 8, 3 (2014), 281–292.

[45] Nan Ma, Jiancheng Guan, and Yi Zhao. 2008. Bringing PageRank to the citation analysis. *Information Processing & Management* 44, 2 (2008), 800–810.

[46] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data.* 135–146.

[47] Fragkiskos D Malliaros, Christos Giatsidis, Apostolos N Papadopoulos, and Michalis Vazirgiannis. 2020. The core decomposition of networks: Theory, algorithms and applications. *The VLDB Journal* 29, 1 (2020), 61–92.

[48] Daniel Mawhirter and Bo Wu. 2019. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 509–523.

[49] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.

[50] Miller McPherson, Lynn Smith-Lovin, and James M Cook. 2001. Birds of a feather: Homophily in social networks. *Annual review of sociology* 27, 1 (2001), 415–444.

[51] Lingkai Meng, Yu Shao, Long Yuan, Longbin Lai, Peng Cheng, Xue Li, Wenyuan Yu, Wenjie Zhang, Xuemin Lin, and Jingren Zhou. 2024. A survey of distributed graph algorithms on massive graphs. *Comput. Surveys* 57, 2 (2024), 1–39.

[52] Eduardo Mosqueira-Rey, David Alonso-Ríos, Vicente Moret-Bonillo, Isaac Fernández-Varela, and Diego Álvarez-Estévez. 2018. A systematic approach to API usability: Taxonomy-derived criteria and a case study. *Inf. Softw. Technol.* 97 (2018), 46–63.

[53] Eduardo Mosqueira-Rey, David Alonso-Ríos, Vicente Moret-Bonillo, Isaac Fernández-Varela, and Diego Álvarez-Estévez. 2018. A systematic approach to API usability: Taxonomy-derived criteria and a case study. *Information and Software Technology* 97 (2018), 46–63.

[54] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19, 45-74 (2010), 22.

[55] Siva Karthik Mustikovela, Michael Ying Yang, and Carsten Rother. 2016. Can ground truth label propagation from video help semantic segmentation?. In *European Conference on Computer Vision*. Springer, 804–820.

[56] Brad A. Myers. 2017. Human-Centered Methods for Improving API Usability. In *1st IEEE/ACM International Workshop on API Usage and Evolution, WAPI@ICSE 2017, Buenos Aires, Argentina, May 23, 2017*. IEEE Computer Society, 2.

[57] Brad A. Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (2016), 62–69.

[58] Daye Nam, Amber Horvath, Andrew Macvean, Brad A. Myers, and Bogdan Vasilescu. 2019. MARBLE: Mining for Boilerplate Code to Identify API Usability Problems. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 615–627.

[59] Danupon Nanongkai. 2014. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. 565–573.

[60] Guang Ouyang, Dipak K Dey, and Panpan Zhang. 2020. Clique-based method for social network clustering. *Journal of Classification* 37, 1 (2020), 254–274.

[61] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford infolab.

[62] Marco Piccioni, Carlo A. Furia, and Bertrand Meyer. 2013. An Empirical Study of API Usability. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013*. IEEE Computer Society, 5–14.

[63] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. 2013. An empirical study of API usability. In *IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 5–14.

[64] Arnau Prat-Pérez and David Dominguez-Sal. 2014. How community-like is the structure of synthetically generated graphs?. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*. 1–9.

[65] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. 2014. Scalable big graph processing in mapreduce. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 827–838.

[66] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 76, 3 (2007), 036106.

[67] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. 2013. Finding connected components in map-reduce in logarithmic rounds. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 50–61.

[68] Irum Rauf, Elena Troubitsyna, and Ivan Porres. 2019. A systematic mapping study of API usability evaluation methods. *Comput. Sci. Rev.* 33 (2019), 49–68.

[69] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 410–424.

[70] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[71] Satu Elisa Schaeffer. 2007. Graph clustering. *Computer science review* 1, 1 (2007), 27–64.

[72] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[73] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.). ACM, 135–146.

[74] Philip Stutz, Abraham Bernstein, and William Cohen. 2010. Signal/collect: graph algorithms for the (semantic) web. In *The Semantic Web–ISWC 2010: 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I 9*. Springer, 764–780.

[75] Robert E Tarjan and Jan Van Leeuwen. 1984. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)* 31, 2 (1984), 245–281.

[76] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 425–440.

[77] AJ Trevor, Suat Gedikli, Radu B Rusu, and Henrik I Christensen. 2013. Efficient organized point cloud segmentation with connected components. *Semantic Perception Mapping and Exploration (SPME)* 1 (2013).

[78] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[79] Paul Vernaza and Manmohan Chandraker. 2017. Learning random-walk label propagation for weakly-supervised semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7158–7166.

[80] Y Shiloachand U Vishkin and Y Shiloach. 1982. An o (log n) parallel connectivityalgorithm. *J. algorithms* 3 (1982), 57–67.

[81] Jia-Ping Wang. 1998. Stochastic relaxation on partitions with connected components and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, 6 (1998), 619–636.

[82] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. 2014. BigDataBench: A big data benchmark suite from internet services. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*. IEEE Computer Society, 488–499.

[83] Qingsi Wang, Xinbing Wang, and Xiaojun Lin. 2009. Mobility increases the connectivity of k-hop clustered wireless networks. In *Proceedings of the 15th annual international conference on Mobile computing and networking*. 121–132.

[84] Xinrui Wang, Yiran Wang, Xuemin Lin, Jeffrey Xu Yu, Hong Gao, Xiuzhen Cheng, and Dongxiao Yu. 2024. Efficient Betweenness Centrality Computation over Large Heterogeneous Information Networks. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3360–3372.

[85] Robert B Watson. 2009. Improving software API usability through text analysis: A case study. In *2009 IEEE International Professional Communication Conference*. IEEE, 1–7.

[86] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440–442.

[87] Christian Wiedemann and Heinrich Ebner. 2000. Automatic completion and evaluation of road networks. *International Archives of Photogrammetry and Remote Sensing* 33, B3/2; PART 3 (2000), 979–986.

[88] Zhihao Wu, Youfang Lin, Jing Wang, and Steve Gregory. 2016. Link prediction with node clustering coefficient. *Physica A: Statistical Mechanics and its Applications* 452 (2016), 1–8.

[89] Feng Xie and David Levinson. 2007. Measuring the structure of road networks. *Geographical analysis* 39, 3 (2007), 336–356.

[90] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, et al. 2017. Big graph analytics platforms. *Foundations and Trends® in Databases* 7, 1-2 (2017), 1–195.

[91] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[92] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*. 1307–1317.

[93] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1821–1832.

[94] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M Tamer Özsu, Wei-Shinn Ku, and John CS Lui. 2020. G-thinker: A distributed framework for mining subgraphs in a big graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1369–1380.

[95] Da Yan, Lyuheng Yuan, Akhlaque Ahmad, Chenguang Zheng, Hongzhi Chen, and James Cheng. 2024. Systems for Scalable Graph Analytics and Machine Learning: Trends and Methods. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 6627–6632.

[96] Xiaoyan Yin, Xiao Hu, Yanjiao Chen, Xu Yuan, and Baochun Li. 2019. Signed-PageRank: An efficient influence maximization framework for signed social networks. *IEEE Transactions on Knowledge and Data Engineering* 33, 5 (2019), 2208–2222.

[97] Zixiao Zhu, Yanzhen Zou, Bing Xie, Yong Jin, Zeqi Lin, and Lu Zhang. 2014. Mining API Usage Examples from Test Code. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 301–310.

Received October 2024; revised January 2025; accepted February 2025