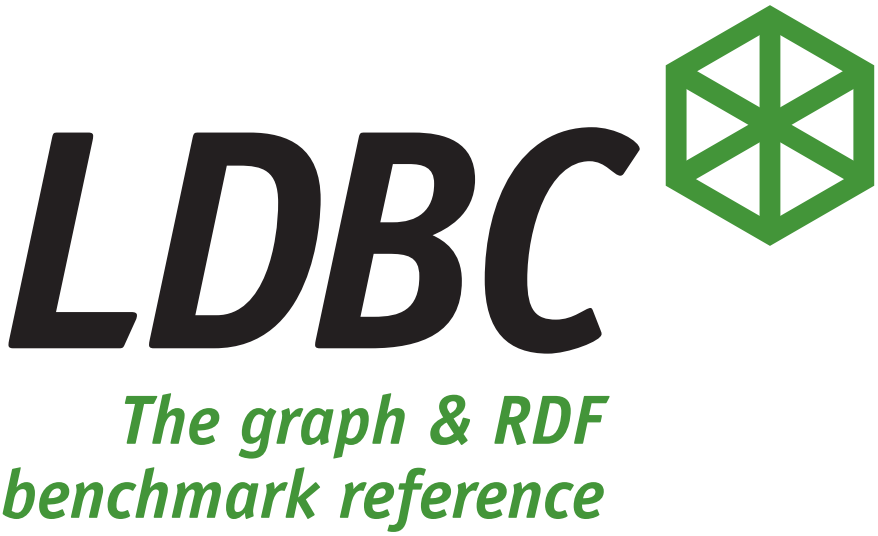


The LDBC Graphalytics Benchmark v1.0.5-draft

Coordinator: Alexandru Iosup (Vrije Universiteit Amsterdam (VU) and Delft University of Technology (TUD))

With contributions from: Ahmed Musaafir (VU), Alexandru Uta (VU), Arnau Prat Pérez (UPC), Gábor Szárnyas (MTA-BME, CWI), Hassan Chafi (Oracle), Ilie Gabriel Tănase (IBM), Lifeng Nai (GT), Michael Anderson (Intel), Mihai Capotă (Intel), Narayanan Sundaram (Intel), Peter Boncz (CWI), Siegfried Depner (formerly Oracle), Stijn Heldens (UTwente, formerly TUD), Thomas Manhardt (Oracle), Tim Hegeman (TUD), Wing Lung Ngai (VU, formerly TUD), Yinglong Xia (Huawei)

The specification was built on the source code available at

`https://github.com/ldbc/ldbc_graphalytics_docs/tree/main`

Abstract

In this document, we describe LDBC Graphalytics, an industrial-grade benchmark for graph analysis platforms. The main goal of Graphalytics is to enable the fair and objective comparison of graph analysis platforms. Due to the diversity of bottlenecks and performance issues such platforms need to address, Graphalytics consists of a set of selected deterministic algorithms for full-graph analysis, standard graph datasets, synthetic dataset generators, and reference output for validation purposes. Its test harness produces deep metrics that quantify multiple kinds of systems scalability, weak and strong, and robustness, such as failures and performance variability. The benchmark also balances comprehensiveness with runtime necessary to obtain the deep metrics. The benchmark comes with open-source software for generating performance data, for validating algorithm results, for monitoring and sharing performance data, and for obtaining the final benchmark result as a standard performance report.

Executive Summary

Processing graphs, especially at large scale, is an increasingly important activity in a variety of business, engineering, and scientific domains. Tens of very different graph processing platforms, such as Giraph, GraphLab, and even generic data processing frameworks such as Hadoop and Spark, can be used for this purpose. For graph processing platforms to be adopted and to continue their evolution, users must be able to select with ease the best-performing graph processing platform, and developers and system integrators have to find it easy to quantify the non-functional aspects of the system, from performance to scalability. Compared to traditional benchmarking, benchmarking graph processing platforms must provide a diverse set of kernels (algorithms), provide recipes for generating diverse yet controlled datasets at large scale, and be portable to diverse and evolving platforms.

The Linked Data Benchmark Council's Graphalytics is a combined industry and academia initiative, formed by principal actors in the field of graph-like data management. The main goal of LDBC Graphalytics is to define a benchmarking framework, and the associated open-source software tools, where different graph processing platforms and core graph data management technologies can be fairly tested and compared. The key feature of Graphalytics is the understanding of the irregular and deep impact that dataset and algorithm diversity can have on performance, leading to bottlenecks and performance issues. For this feature, Graphalytics proposes a set of selected deterministic algorithms for full-graph analysis, standard graph datasets, synthetic dataset generators, and reference output for validation purposes. Furthermore, the Graphalytics test harness can be used to conduct diverse experiments and produce deep metrics that quantify multiple kinds of systems scalability, weak and strong, and robustness, such as failures and performance variability. The benchmark also balances comprehensiveness with runtime necessary to obtain the deep metrics. Because issues change over time, LDBC Graphalytics also proposes a renewal process.

Overall, Graphalytics aims to drive not only the selection of adequate graph processing platforms, but also help with system tuning by providing data for system-bottleneck identification, and with feature and even system (re-)design by providing quantitative evidence of the presence of sub-optimal designs. To this end, the development of the benchmark follows and extends the guidelines set by other LDBC benchmarks, such as the LDBC Social Network Benchmark's Interactive and Business Intelligence workloads.

To increase adoption by industry and research organizations, LDBC Graphalytics provides all the necessary software to run its comprehensive benchmark process. The open-source software contains tools for generating performance data, for validating algorithm results, and or monitoring and sharing performance data. The software is designed to be easy to use and deploy at a small cost. Last, the software is developed using modern software engineering practices, with low technical debt, high quality of code, and deep testing and validation processes prior to release.

This preliminary version of the LDBC Graphalytics specification contains a formal definition of the benchmarking framework and of its components, a description of the benchmarking process, links to the open-source software including a working benchmarking harness and drivers for several vendor- and community-driven graph processing platforms, pseudo-code for all algorithms included in the benchmark, and a comparison with related tools, products, and concepts in the benchmarking space.

Table of Contents

List of Tables

In this work we introduce the LDBC Graphalytics benchmark, explaining the motivation for creating this benchmark suite for graph analytics platforms, its relevance to industry and academia, the overview of the benchmark process, and the participation of industry and academia in developing this benchmark. A scientific companion to this technical specification has been published in 2016 [30].

1.1 Motivation for the Benchmark

Responding to increasingly larger and more diverse graphs, and the need to analyze them, both industry and academia are developing and tuning graph analysis software platforms. Already tens of such platforms exist, but their performance is often difficult to compare. Moreover, the random, skewed, and correlated access patterns of graph analysis, caused by the complex interaction between input datasets and applications processing them, expose new bottlenecks on the hardware level, as hinted at by the large differences between Top500 and Graph500 rankings (see Appendix C for the related work). Therefore, addressing the need for fair, comprehensive, standardized comparison of graph analysis platforms, in this work we propose the LDBC Graphalytics benchmark.

1.2 Relevance to Industry and Academia

A standardized, comprehensive benchmark for graph analytics platforms is beneficial to both industry and academia. Graphalytics allows a comprehensive, fair comparison across graph analysis platforms. The benchmark results provide insightful knowledge to users and developers on performance tuning of graph processing, and increases the understanding of the advantages and disadvantages of the design and implementation, therefore stimulating academic research in graph data storage, indexing, and analysis. By supporting platform variety, it reduces the learning curve of new users to graph processing systems.

1.3 General Benchmark Overview

This benchmark suite evaluates the performance of graph analysis platforms that facilitate complex and holistic graph computations. This benchmark must comply to the following requirements: (R1) targeting platforms and systems; (R2) incorporating diverse, representative benchmark elements; (R3) using a diverse, representative process; (R4) including a renewal process; (R5) developed under modern software engineering techniques.

In the benchmark (see Chapter 2 for the formal definition), we carefully motivate the choice of our algorithms and datasets to conduct our benchmark experiments. Graphalytics consists of six core algorithms (also known as kernels [4]): breadth-first search, PageRank, weakly connected components, community detection using label propagation, local clustering coefficient, and single-source shortest paths. The workload includes real and synthetic datasets, which are classified into intuitive "T-shirt" sizes (e.g., S, M, L, XL). The benchmarking process is made future-proof, through a *renewal process*.

Each system under test undergoes a standard benchmark (see Chapter 3) per target scale, which executes in total 90 graph-processing jobs (six core algorithms, five different datasets, and three repetitions per job).

1.4 Participation of Industry and Academia

The Linked Data Benchmark Council (`ldbcouncil.org`, LDBC), is an industry council formed to establish standard benchmark specifications, practices and results for *graph data management systems*. The list of institutions that take part in the definition and development of LDBC Graphalytics is formed by relevant actors from both the industry and academia in the field of large-scale graph processing. As of February 2017, the list of participants is as follows:

- CENTRUM WISKUNDE & INFORMATICA, the Netherlands (CWI)
- DELFT UNIVERSITY OF TECHNOLOGY, the Netherlands (TUD)
- VRIJE UNIVRSITEIT AMSTERDAM, the Netherlands (VU)
- GEORGIA INSTITUTE OF TECHNOLOGY, USA (GT)
- HUAWEI RESEARCH AMERICA, USA (HUAWEI)
- INTEL LABS, USA (INTEL)
- ORACLE LABS, USA (ORACLE)
- POLYTECHNIC UNIVERSITY OF CATALONIA, Spain (UPC)
- MTA-BME LENDÜLET RESEARCH GROUP ON CYBER-PHYSICAL SYSTEMS AT THE BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS, Hungary (MTA-BME)

1.5 Technical report

This technical report is available on arXiv [31].

2.1 Requirements

The Graphalytics benchmark is the result of a number of design choices.

(R1) Target platforms and systems: benchmarks must support any graph analysis platform operating on any underlying hardware system. For platforms, we do not distinguish between programming models and support any model. For systems, we target the following environments: multi-core and many-core single-node systems, systems with accelerators (GPUs, FPGAs, ASICs), hybrid systems, and distributed systems that possibly combine several of the previous types of environments. Without R1, a benchmark could not service a diverse industrial following.

(R2) Diverse, representative benchmark elements: data model and workload selection must be representative and have a good coverage of real-world practice. In particular, the workload selection must include datasets and algorithms which cover known system bottlenecks and be representative in the current and near-future practice. Without representativeness, a benchmark could bias work on platforms and systems towards goals that are simply not useful for improving current practice. Without coverage, a benchmark could push the community into pursuing cases that are currently interesting for industry, but not address what could become impassable bottlenecks in the near-future.

(R3) Diverse, representative process: the set of experiments conducted by the benchmark automatically must be broad, covering the main bottlenecks of the target systems. The process must also include validation of results, thus making sure the processing is done correctly. Without R3, a benchmark could test very few of the diverse capabilities of the target platforms and systems, and benchmarking results could not be trusted.

(R4) Include renewal process: unlike many other benchmarks, benchmarks in the area of graph processing must include a renewal process, that is, not only a mechanism to scale up or otherwise change the workload to keep up with increasing more powerful systems, but also a process to automatically configure the mechanism, and a way to characterize the reasonable characteristics of the workload for an average platform running on an average system. Without R4, a benchmark could become less relevant for the systems of the future.

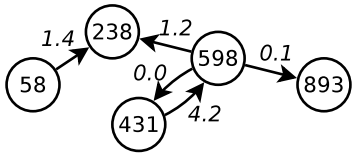
(R5) Modern software engineering: benchmarks must include a modern software architecture and run a modern software-engineering process. The Graphalytics benchmark is provided with an extensive benchmarking suite that allows users to easily add new platforms and systems to test. This makes it possible for practitioners to easily access the benchmarks and compare their platforms and systems against those of others. Without R5, a benchmark could easily become unmaintainable or unusable.

2.2 Data

The Graphalytics benchmark operates on a single type of dataset: graphs. This section provides the definition of a graph, the definition of the representation used by Graphalytics for its input and output data, and the datasets used for the benchmarks.

2.2.1 Definition

Graphalytics does not impose any requirements on the semantics of the graph. The benchmark uses a typical data model for graphs. A graph consists of a collection of *vertices* (nodes) which are linked by *edges* (relationships). Each vertex is assigned a unique identifier represented as an unsigned 64-bit integer, i.e., between 0 and $2^{64} - 1$. Vertex identifiers do not necessarily start at zero, nor are the identifiers necessarily consecutive. Edges



(a) Graph

```
58
238
431
598
893
```

(b) Vertex file

```
58 238 1.4
431 598 4.2
598 238 1.2
598 431 0.0
598 893 0.1
```

(c) Edge file

Figure 2.1: Example of directed weighted graph in EVLP format.

are represented as pairs of vertex identifiers. Graphalytics supports both *directed* graphs (i.e., edges are unidirectional) and *undirected* graphs (i.e., edges are bidirectional). Every edge is unique and connects two distinct vertices. This implies that self-loops (i.e., vertices having edges to themselves) and multi-edges (i.e., multiple edges between a pair of vertices) are not allowed. In the case of undirected graphs, for every pair of vertices u and v, the edges (u, v) and (v, u) are considered to be identical.

Vertices and edges can have properties which can be used to store meta-data such as weights, timestamps, labels, costs, or durations. Currently, Graphalytics supports three types of properties: *integers*, *floating-point numbers*, and *booleans*. All floating-point numbers must be internally stored and handled in 64-bit double-precision IEEE 754 format. We explicitly do not allow the single precision format, as this can speedup computation and presents an unfair advantage. The current specification of Graphalytics does not use integer or boolean properties.

2.2.2 Representation

In Graphalytics, the file format used to represent the graphs is the "Edge/Vertex-List with Properties" (*EVLP*) format for graphs. The format consists of two text files: a vertex-file containing the vertices (with optional properties) and an edge-list containing the edges (with optional properties). Both files are plain text (ASCII) and consist of a sequence of lines.

For the vertex files, each line contains exactly one vertex identifier. The vertex identifiers are sorted in ascending order to facilitate easy conversion to other formats. For the edge files, each line contains two vertex identifiers separated by a space. The edges are sorted in lexicographical order based on the two vertices. For directed graphs, the source vertex is listed first. For undirected graphs, the smallest identifier of the two vertices is listed first and each edge is only listed once in one direction.

Vertices and edges can have optional properties. These values of these properties are listed in the vertex/edge files after each vertex/edge and are separated by spaces. The interpretation of these properties is not provided by the files.

Figure 2.1 shows an example of the EVLP format for a small directed weighted graph consisting of 5 vertices and 5 directed edges.

2.2.3 Size and Scale

Graphalytics includes both graphs from real-world applications and synthetic graphs which are created using graph generators. Graphalytics uses a broad range of graphs with a large variety in domain, size, density, and other characteristics. To facilitate performance comparison across datasets, the *scale* of a graph is derived from the number of vertices (n) and the number of edges (m). Formally, the scale of a graph is defined by calculating the sum $n + m$, taking the logarithm of this value in base 10, and truncating the result to one decimal place. Formally, this can be written as follows:

$$\mathit{Scale}(n, m) = \lfloor 10 \log_{10}(n + m) \rfloor / 10 \tag{2.1}$$

The scale of a graph gives users an intuition of what the size of graph means in practice. Scales are grouped into classes spanning 0.5 *scale units* and these classes are labeled using familiar system of "T-shirt sizes": small (S),

Label	Scales
2XS	$6.5 - 6.9$
XS	$7.0 - 7.4$
S	$7.5 - 7.9$
M	$8.0 - 8.4$
L	$8.5 - 8.9$
XL	$9.0 - 9.4$
2XL	$9.5 - 9.9$
3XL	$10.0 - 10.4$

Table 2.1: Mapping of dataset scales ("T-shirt sizes") in Graphalytics.

medium (M), and large (L), with extra (X) prepended for extremes scales. The reference point is class L, which is defined by the Graphalytics team as the largest class such that the BFS algorithm completes within an hour on any graph from that scale using a state-of-the-art graph analysis platform and a single commodity machine. Table 2.1 summarizes the classes used in Graphalytics.

2.2.4 Datasets

Graphalytics uses both graphs from real-world applications and synthetic graphs which are generated using data generators, the selection of which spans a variety of sizes and densities.

The Graphalytics data sets are available in the SURF/CWI data repository [29] at `https://repository.surfsara.nl/datasets/cwi/graphalytics`.

Real-world Datasets By including real-world graphs from a variety of domains, Graphalytics covers users from different communities, including graphs from the knowledge, gaming, and social network domains. Table 2.2 lists the real-world datasets used by the Graphalytics benchmark.

ID	Name	n	m	Scale	Domain
R1(2XS)	wiki-talk [34]	2.39 M	5.02 M	6.9	Knowledge
R2(XS)	kgs [22]	0.83 M	17.9 M	7.3	Gaming
R3(XS)	cit-patents [34]	3.77 M	16.5 M	7.3	Knowledge
R4(S)	dota-league [22]	0.06 M	50.9 M	7.7	Gaming
R5(XL)	com-friendster [34]	65.6 M	1.81 B	9.3	Social
R6(XL)	twitter_mpi [8]	52.6 M	1.97 B	9.3	Social

Table 2.2: Real-world datasets used by Graphalytics.

Synthetic Datasets Besides real-world datasets, Graphalytics adopts two commonly used generators that generate two types of graphs: power-law graphs from the **Graph500** generator [9, 37] and social network graphs from **LDBC Datagen** [17]. Tables 2.3 and 2.4 list the Graph500 datasets and the Datagen datasets.

2.3 Algorithms

The Graphalytics benchmark consists of six algorithms (also known as *kernels* [4]) which need to be executed on the different datasets: five algorithms for unweighted graphs and one algorithm for weighted graphs. These algorithms have been selected based on the results of multiple surveys and expert advice from the participants of the LDBC Technical User Community (TUC) meeting.

ID	Name	n	m	Scale
G22(S)	Graph500-22	2.4M	64.2M	7.8
G23(M)	Graph500-23	4.6M	129.3M	8.1
G24(M)	Graph500-24	8.9M	260.4M	8.4
G25(L)	Graph500-25	17.0M	523.6M	8.7
G26(XL)	Graph500-26	32.8M	1.1B	9.0
G27(XL)	Graph500-27	65.6M	2.1B	9.3
G28(2XL)	Graph500-28	121M	4.2B	9.6
G29(2XL)	Graph500-29	233M	8.5B	9.9
G30(3XL)	Graph500-30	448M	17.0B	10.2

Table 2.3: Synthetic Graph500 datasets used by Graphalytics.

ID	Name	n	m	Scale
D7.5(S)	Datagen-7.5-fb	0.6M	34.2M	7.5
D7.6(S)	Datagen-7.6-fb	0.8M	42.2M	7.6
D7.7(S)	Datagen-7.7-zf	13.2M	32.8M	7.6
D7.8(S)	Datagen-7.8-zf	16.5M	41.0M	7.7
D7.9(S)	Datagen-7.9-fb	1.4M	85.7M	7.9
D8.0(M)	Datagen-8.0-fb	1.7M	107.5M	8.0
D8.1(M)	Datagen-8.1-fb	2.1M	134.3M	8.1
D8.2(M)	Datagen-8.2-zf	43.7M	106.4M	8.1
D8.3(M)	Datagen-8.3-zf	53.5M	130.6M	8.2
D8.4(M)	Datagen-8.4-fb	3.8M	269.5M	8.4
D8.5(L)	Datagen-8.5-fb	4.6M	332.0M	8.5
D8.6(L)	Datagen-8.6-fb	5.7M	422.0M	8.6
D8.7(L)	Datagen-8.7-zf	145.1M	340.2M	8.6
D8.8(L)	Datagen-8.8-zf	168.3M	413.4M	8.7
D8.9(L)	Datagen-8.9-fb	10.6M	848.7M	8.9
D9.0(XL)	Datagen-9.0-fb	12.9M	1.0B	9.0
D9.1(XL)	Datagen-9.1-fb	16.1M	1.3B	9.1
D9.2(XL)	Datagen-9.2-zf	434.9M	1.0B	9.1
D9.3(XL)	Datagen-9.3-zf	555.3M	13.1B	9.2
D9.4(XL)	Datagen-9.4-fb	29.3M	2.6B	9.4
D-3k(XL)	Datagen-sf3k-fb	33.5M	2.9B	9.4
D-10k(2XL)	Datagen-sf10k-fb	100.2M	9.4B	9.9

Table 2.4: Synthetic Datagen datasets used by Graphalytics.

Each workload of Graphalytics consists of executing a single algorithm on a single dataset. Below, abstract descriptions are provided for the six algorithms; pseudo-code is given in Appendix A. Furthermore, a link to the reference implementation is presented in **??**. However, Graphalytics does not impose any constraint on the implementation of algorithms. Any implementation is allowed, as long as its correctness can be validated by comparing its output to correct reference output (Section 2.4).

In the following sections, a graph G consists of a set of vertices V and a set of edges E. For undirected graphs, each edge is bidirectional, so if $(u, v) \in E$ then $(v, u) \in E$. Each vertex v has a set of outgoing neighbors $N_{\text{out}}(v) = \{u \in V | (v, u) \in E\}$, a set of incoming neighbors $N_{\text{in}}(v) = \{u \in V | (u, v) \in E\}$, and a set of all neighbors $N(v) = N_{\text{in}}(v) \cup N_{\text{out}}(u)$. Note that for undirected graphs, each edge is bidirectional so we have $N_{\text{in}}(v) = N_{\text{out}}(v) = N(v)$.

2.3.1 Breadth-First Search (BFS)

Breadth-First Search is a traversal algorithm that labels each vertex of a graph with the length (or *depth*) of the shortest path from a given source vertex (*root*) to this vertex. The root has depth 0, its outgoing neighbors have depth 1, their outgoing neighbors have depth 2, etc. Unreachable vertices should be given the value infinity (represented as `9223372036854775807`). Example graphs are shown in Figure D.1.

2.3.2 PageRank (PR)

PageRank is an iterative algorithm that assigns to each vertex a ranking value. The algorithm was originally used by Google Search to rank websites in their search results [41]. Let $PR_i(v)$ be the PageRank value of vertex v after iteration i. Initially, each vertex v is assigned the same value such that the sum of all vertex values is 1.

$$PR_0(v) = \frac{1}{|V|} \tag{2.2}$$

After iteration i, each vertex pushes its PageRank over its outgoing edges to its neighbors. The PageRank for each vertex is updated according to the following rule:

$$PR_i(v) = \underbrace{\frac{1-d}{|V|}}_{\textit{teleport}} + \underbrace{d \cdot \sum_{u \in N_{\text{in}}(v)} \frac{PR_{i-1}(u)}{|N_{\text{out}}(u)|}}_{\textit{importance}} + \underbrace{\frac{d}{|V|} \cdot \sum_{w \in D} PR_{i-1}(w)}_{\textit{redistributed from sinks}} \tag{2.3}$$

Notation: $d \in [0, 1]$ is the *damping factor* and $D = \left\{ w \in V \;\middle|\; |N_{\text{out}}(w)| = 0 \right\}$ is the set of *sink vertices*, i.e., vertices having no outgoing edges. Sink vertices have nowhere to push their PageRank to, so the total sum of the PageRanks for the sink vertices is evenly distributed over all vertices [14]. When computing the *importance* value, we consider $\frac{PR_{i-1}(u)}{|N_{\text{out}}(u)|}$ to be 0 for sink vertices.

The PageRank algorithm should continue for a fixed number of iterations. The floating-point values must be handled as 64-bit double-precision IEEE 754 floating-point numbers.

2.3.3 Weakly Connected Components (WCC)

This algorithm finds the *Weakly Connected Components* of a graph and assigns each vertex a unique label that indicates which component it belongs to. Two vertices belong to the same component, and thus have the same label, if there exists a path between these vertices along the edges of the graph. For directed graphs, it is allowed to travel over the reverse direction of an edge, i.e., the graph is interpreted as if it is undirected. Example graphs are shown in Figure D.6.

2.3.4 Community Detection using Label Propagation (CDLP)

The *community detection* algorithm in Graphalytics uses *label propagation* (CDLP) and is based on the algorithm proposed by Raghavan et al. [43]. The algorithm assigns each vertex a label, indicating its community, and these labels are iteratively updated where each vertex is assigned a new label based on the frequency of the labels of its neighbors. The original algorithm has been adapted to be both deterministic and parallel, thus enabling output validation and parallel execution.

Let $L_i(v)$ be the label of vertex v after iteration i. Initially, each vertex v is assigned a unique label which matches its identifier.

$$L_0(v) = v \tag{2.4}$$

In iteration i, each vertex v determines the frequency of the labels of its incoming and outgoing neighbors and selects the label which is most common. If the graph is directed and a neighbor is reachable via both an incoming and outgoing edge, its label will be counted twice. In case there are multiple labels with the maximum

frequency, the smallest label is chosen. In case a vertex has no neighbors, it retains its current label. This rule can be written as follows:

$$L_i(v) = \min\left(\arg\max_l \left[\left|\{u \in N_{\text{in}}(v) \mid L_{i-1}(u) = l\}\right| + \left|\{u \in N_{\text{out}}(v) \mid L_{i-1}(u) = l\}\right|\right]\right) \tag{2.5}$$

Example graphs are shown in Figure D.2.

Note. The CDLP algorithm in Graphalytics has two key differences from the original algorithm proposed in [43]. First, it is deterministic: if there are multiple labels with their frequency equalling the maximum, it selects the smallest one while the original algorithm selects randomly. Second, it is synchronous, i.e., each iteration is computed based on the labels obtained as a result of the previous iteration. As remarked in [43], this can cause the oscillation of labels in bipartite or nearly bipartite subgraphs.

2.3.5 Local Clustering Coefficient (LCC)

The *Local Clustering Coefficient* algorithm determines the local clustering coefficient for each vertex. This coefficient indicates the ratio between the number of edges between the neighbors of a vertex and the maximum number of possible edges between the neighbors of this vertex. If the number of neighbors of a vertex is less than two, its coefficient is defined as zero. The definition of LCC can be written as follows:

$$LCC(v) = \begin{cases} 0 & \text{If } |N(v)| \leq 1 \\ \frac{|\{(u,w)|u,w \in N(v) \wedge (u,w) \in E\}|}{|(u,w)|u,w \in N(v)|} & \text{Otherwise} \end{cases} \tag{2.6}$$

Note that the second case can also be written using the sum over the neighbors of v.

$$LCC(v) = \frac{\sum_{u \in N(v)} |N(v) \cap N_{\text{out}}(u)|}{|N(v)|\big(|N(v)| - 1\big)} \tag{2.7}$$

For *directed graphs*, the set of neighbors $N(v)$ is determined without taking directions into account, but each neighbor is only counted once (a neighbor with both an incoming and an outgoing edge from vertex v does not count twice). However, directions are enforced when determining $N_{\text{out}}(v)$ between neighbors. Note that calculating the intersection using the *incoming* edges to u yields the same result, i.e.

$$\sum_{u \in N(v)} |N(v) \cap N_{\text{out}}(u)| = \sum_{u \in N(v)} |N(v) \cap N_{\text{in}}(u)| \tag{2.8}$$

Example graphs are shown in Figure D.3 and Figure D.4.

2.3.6 Single-Source Shortest Paths (SSSP)

The *Single-Source Shortest Paths* algorithm marks each vertex with the length of the shortest path from a given *root* vertex to every other vertex in the graph. The length of a path is defined as the sum of the weights on the edges of the path. The edge weights are floating-point numbers which must be handled as 64-bit double-precision IEEE 754 floating-point numbers. The edge weights are never negative, infinity, or invalid (i.e., *NaN*), but are allowed to be zero. Unreachable vertices should be given the value infinity (represented as `infinity`).

Example graphs are shown in Figure D.5.

2.4 Output Validation

The output of every execution of an algorithm on a dataset must be validated for the result to be admissible. All algorithms in the Graphalytics benchmark are deterministic and can therefore be validated by comparing to reference output for correctness. The reference output is typically generated by a specifically chosen reference

platform, the implementation of which is cross-validated with at least two other platforms up to target scale L. The results are tested by cross-validating multiple platforms and implementations against each others (see **??**).

The validation output presents numbers either as integers or floating-point numbers, depending on the algorithm definition. Note that these numbers are stored in the file system as decimal values in plain text (ASCII). For floating-point numbers, a scientific notation with 15 significant digits (e.g., $2.476\,533\,217\,845\,853\mathrm{e}{-08}$) is used.

The system's output generated during the benchmark must be stored as decimal values in plain text (ASCII) files, grouped into a single file directory. The formatting rules for integers must be exactly the same as the validation output.

There are three methods used for validation:

```
1 3
2 1
3 2
4 0
5 1
```

(a) Reference output

```
1 3
2 1
3 2
4 0
5 1
```

(b) Example of correct result

```
1 4
2 1
3 2
4 5
5 1
```

(c) Example of incorrect result

Figure 2.2: Example of validation with *exact match*.

```
1 1
2 1
3 1
4 2
5 2
6 3
```

(a) Reference output

```
1 81
2 81
3 81
4 32
5 32
6 12
```

(b) Example of correct result

```
1 31
2 52
3 31
4 31
5 31
6 74
```

(c) Example of incorrect result

Figure 2.3: Example of validation with *equivalence match*.

```
1 0
2 0.3
3 0.45
4 0.23
5 9223372036854775807
6 0.001
```

(a) Reference output

```
1 0
2 0.30002
3 0.45
4 0.229997
5 9223372036854775807
6 0.001
```

(b) Example of correct result

```
1 0.000001
2 0.3
3 0.46
4 0.22
5 1.79769e+308
6 0
```

(c) Example of incorrect result

Figure 2.4: Example of validation with *epsilon match*.

- **Exact match (applies to BFS, CDLP):** the vertex values of the system's output should be identical to the reference output. Figure 2.2 shows an example of validation with exact match.
- **Equivalence match (applies to WCC):** the vertex values of the system's output should be equal to the reference output *under equivalence*. This means a two-way mapping should exists that maps the system's

output to be identical to reference output and the inverse of this mapping maps the reference output to be identical to the system's output. In other words, the output is considered to be valid if all vertices which have the same label in the system's output also have the same label in the reference output, and vice versa. Figure 2.3 shows an example of validation with equivalence.

- **Epsilon match (applies to PR, LCC, SSSP):** a margin of error is allowed for some algorithms due to floating-point rounding errors. Let r be the reference value of a vertex and s be the system's output value of the same vertex. These values are considered to match if s is within 0.01% of r, i.e., the equation $|r - s| \leq \varepsilon |r|$ holds where $\varepsilon = 0.0001$ (equality is allowed such that the case where $r = 0$ and $s = 0$ passes). The value of ε was chosen such that errors that result from rounding are not penalized. Figure 2.4 shows an example of validation with epsilon match.

Small validation example graphs are available in the Graphalytics suite and are listed in Appendix D, including a common validation graph for all six algorithms (Figure D.7).

2.5 Job

A graph-processing job is the process of executing a graph algorithm (see Section 2.3) on a graph dataset (see Section 2.2.4). This section discusses the description of a graph-processing job, the underlying operations constituting a graph processing job, and the metrics used to measure the job performance.

2.5.1 Description

Each graph-processing job is specified by a list of descriptive information, specifically, the system description, the algorithm, the dataset, and the benchmark configuration. The job description should be uniformly applicable to any graph-processing system.

- **System**: platform type, environment type, and system cost.
- **Algorithm**: algorithm type, and algorithm parameters.
- **Dataset**: vertex size, edge size, and graph size (vertex size + edge size).
- **Benchmark**: deployment mode, allocated resources, and time-out duration.

2.5.2 Operations

Graph processing is data-intensive, sensitive to the data irregularity, and often involves iterative processing. Typically, a graph-processing system facilitates a **Loading** phase to pre-process the data, and follows with one or more **Running** phases to run various graph algorithms on the pre-processed data.

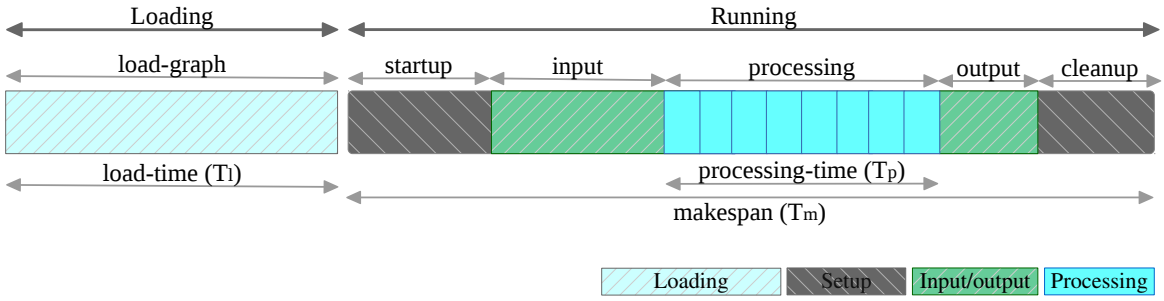


Figure 2.5: A typical graph processing job with the underlying operations.

During the **Loading** step, the input graph data can be converted into an optimized system-specific data format and pre-loaded into a local/share/distributed storage system. Binary formats are allowed, however, the

pre-processed data must be uniformly usable for any graph algorithms targeting the same graph type (e.g., algorithms on unweighted graphs must use the same binary format).

During the **Running** step, the graph-processing system carries out a series of operations to facilitate efficient algorithm execution on the pre-processed data. These operations are categorized into three types: setup, input/output, and processing operations (see Figure 2.5).

- **Setup operations** reserve computational resources in distributed environments, prepare the system for operation, clean up the environment after the termination of the job.
- **Input/output operations** transfer graph data from storage to the memory space, and convert the data to specific formats before/after data processing, and offload the outputs back to the storage. Some platforms load/unload data from a distributed file system, other distributed platforms from share/local storage in each node.
- **Processing operations** take in-memory data and process it according to an user-defined algorithm and its expression in a programming paradigm. Processing operations typically include iterative "processing" steps.

2.5.3 Metrics

This section describes the metrics used in Graphalytics. The Graphalytics benchmark includes several metrics to quantify the performance and other characteristics of the system under test. The performance of graph analytics systems is measured by the time spent on several phases in the execution of the benchmark. Graphalytics reports performance, throughput metrics, cost metrics, and ratio metrics, as follows.

The **Performance metrics** report the execution time of various platform operations.

- **Load time** (T_l), in *seconds*: The time spent loading a particular graph into the system under test, including any preprocessing to convert the input graph to a format suitable for the system. This phase is executed once per graph before any commands are issued to execute specific algorithms.
- **Makespan:** (T_m), in *seconds*: The time between the Graphalytics driver issuing the command to execute an algorithm on a (previously uploaded) graph and the output of the algorithm being made available to the driver. The makespan can be further divided into processing time and overhead. The makespan metric corresponds to the operation of a *cold graph-processing system*, which depicts the situation where the system is started up, processes a single dataset using a single algorithm, and then is shut down.
- **Processing time** (T_p), in *seconds*: Time required to execute an actual algorithm. This does not include platform-specific overhead, such as allocating resources, loading the graph from the file system, or graph partitioning. The processing time metric corresponds to the operation of an in-production, *warmed-up graph-processing system*, where especially loading of the graph from the file system and graph partitioning, both of which are typically done only once and are algorithm-independent, are not considered.

The execution time is capped by the time-out duration configured in each benchmark. Once the time-out is reached, the graph-processing job is terminated, and the time-out duration is reported as the performance metrics instead.

The **Throughput metrics** focus on the processing rate of the system under test. They use a notion of workload intensity, expressed in the graph-specific number of processed edges and vertices:

- **Edges Per Second** (*EPS*), in *units*$/$*second*: The ratio between the number of edges processed (edge size) and the processing time (T_p) is used by other benchmarks (Graph500 in Table C.1) to quantify the rate of operation of the system under test. This is fine for edge-dominated algorithms, such as the BFS used in the same benchmarks, but does not explain the performance of vertex-dominated algorithms or of algorithms whose performance is a complex function of the structural properties of the dataset.

- **Edges and Vertices per Second** (*EVPS*), in *units/second*: Graphalytics uses the ratio between the sum of the number of edges and the number of vertices (graph size) processed by the system, and the processing time (T_p). EVPS is closely related to the scale of a graph, as defined by Graphalytics (see Section 2.2.3).

Graphalytics also reports **Cost metrics**:

- **Three-year Total Cost of Ownership** (*TCO*), in *dollars*: reported in compliance with the LDBC rules [33], so *not* computed by Graphalytics. In particular, LDBC currently adapts the TPC standard pricing model v2.0.0 [50].
- **Price-per-performance** (*PPP*), in *dollars/unit*: as the ratio between TCO and EVPS. This is a metric included for compliance with the LDBC charter [33].

2.6 System Under Test

Responding to requirement R1 (see Section 2.1), the LDBC Graphalytics framework defines the System Under Test as the combined software platform and hardware environment that is able to execute graph-processing algorithms on graph datasets. This is an inclusive definition, and indeed Graphalytics has been executed in the lab of SUTs with software ranging from community-driven prototype systems, to vendor-optimized software; and with hardware ranging from beefy single-node multi-core systems, to single-node CPU and (multi-)GPU hybrid systems, to multi-node clusters with or without GPUs present.

2.7 Renewal Process

To ensure the relevance of Graphalytics as a benchmark for future graph analytics systems, a renewal process is included in Graphalytics. This renewal process updates the workload of the benchmark to keep it relevant for increasingly powerful systems and developments in the graph analytics community. This results in a benchmark which is future-proof. Renewing the benchmark means renewing the algorithms as well as the datasets. For every new version of Graphalytics, a two-stage selection process will be used by the LDBC Graphalytics Task Force. The same selection process was used to derive the workload in the current version of the Graphalytics benchmark.

To achieve both workload representativeness and workload coverage, a two-stage selection process is used. The first stage identifies classes of algorithms and datasets that are representative for real-world usage of graph analytics systems. In the second stage, algorithms and datasets are selected from the most common classes such that the resulting selection is diverse, i.e., the algorithms cover a variety of computation and communication patterns, and the datasets cover a range of sizes and a variety of graph characteristics.

Updated versions of the Graphalytics benchmark will also include renewed definitions of the scale classes defined in Section 2.2.3. The definition of the scale classes is derived from the capabilities of state-of-the-art graph analytics systems and common-off-the-shelf machines, which are expected to be improved over time. Thus, graphs that are considered to be large as of the publication of the first edition of Graphalytics (labeled $L'16$ to indicate the 2016 edition) may be considered medium-sized graphs in the next edition (e.g., $M'20$).

The Graphalytics benchmark suite is developed to facilitate the benchmark process described in this technical specification. This chapter describes the benchmark composition, the benchmark type, the detailed steps of the benchmark execution, and the format of the benchmark report.

3.1 Benchmark

A benchmark is a standardized process to quantify the performance of the system under test. Figure 3.1 depicts the benchmark composition: each benchmark contains a set of benchmark experiments, each experiment consists of multiple benchmark jobs, and each job is executed repeatedly in the form of benchmark runs.

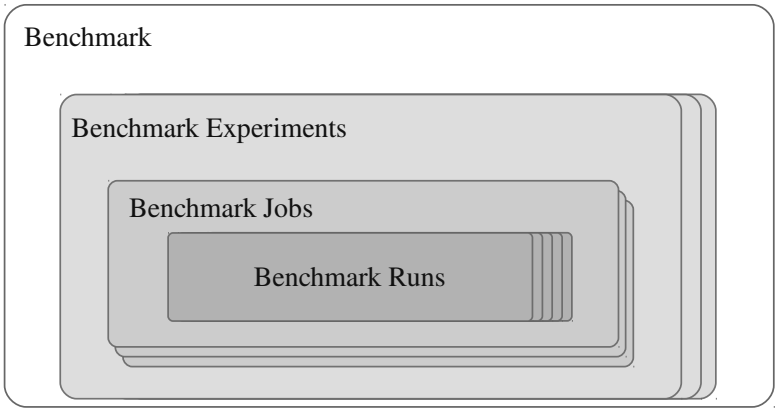


Figure 3.1: The composition of a benchmark.

- A **benchmark experiment** addresses a specific performance characteristic of the system under test, e.g., the performance of an algorithm, or the weak scalability of a system. Each experiment gathers benchmark results from multiple benchmark jobs to quantify a specific performance characteristic.
- A **benchmark job** describes, uniformly across all system under tests, the exact job specification of a graph-processing job (see Section 2.5). The job specification contains information, e.g., the system under test (the platform and the environment), the type of algorithm and dataset, and how much resources are used. These information instructs how the system should be configured during the benchmark execution.
- A **benchmark run** is a real-world execution of a benchmark job. To gather statistically reliable benchmark results, each benchmark job is repeated multiple times in the form of a benchmark run to mitigate the performance variability during the benchmark execution.

3.2 Benchmark Type

The Graphalytics benchmark suite supports four types of benchmark: *test*, *standard*, *full*, and *custom*. This section describes the differences and the composition of these four benchmark types.

3.2.1 Competition Benchmark

Participating in the Graphalytics competition executes the benchmark for ecah data set size category.

The competition benchmark evaluates the system performance with six core algorithms, BFS, WCC, PR, CDLP, LCC, SSSP (see Section 2.3). For each algorithm, the datasets within a given size category are used. A competition benchmark can fall into one of the five target scales: S, M, L, XL, and 2XL+. Each target scale

Size	Timeout
S	15 minutes
M	30 minutes
L	1 hour
XL	2 hours
2XL+	3 hours

Table 3.1: Benchmarks used in the competition

focuses on processing graphs within certain range of data size, and therefore a corresponding time-out duration has been imposed as shown in Table 3.1.

Each algorithm is executed 3 times.

3.3 Benchmark Execution

The benchmark execution of Graphalytics benchmark suite is illustrated in Figure 3.2. This section explains how the benchmark suite executes a benchmark with regard to its execution flow, run flow, data flow, metric collection, and failure indication.

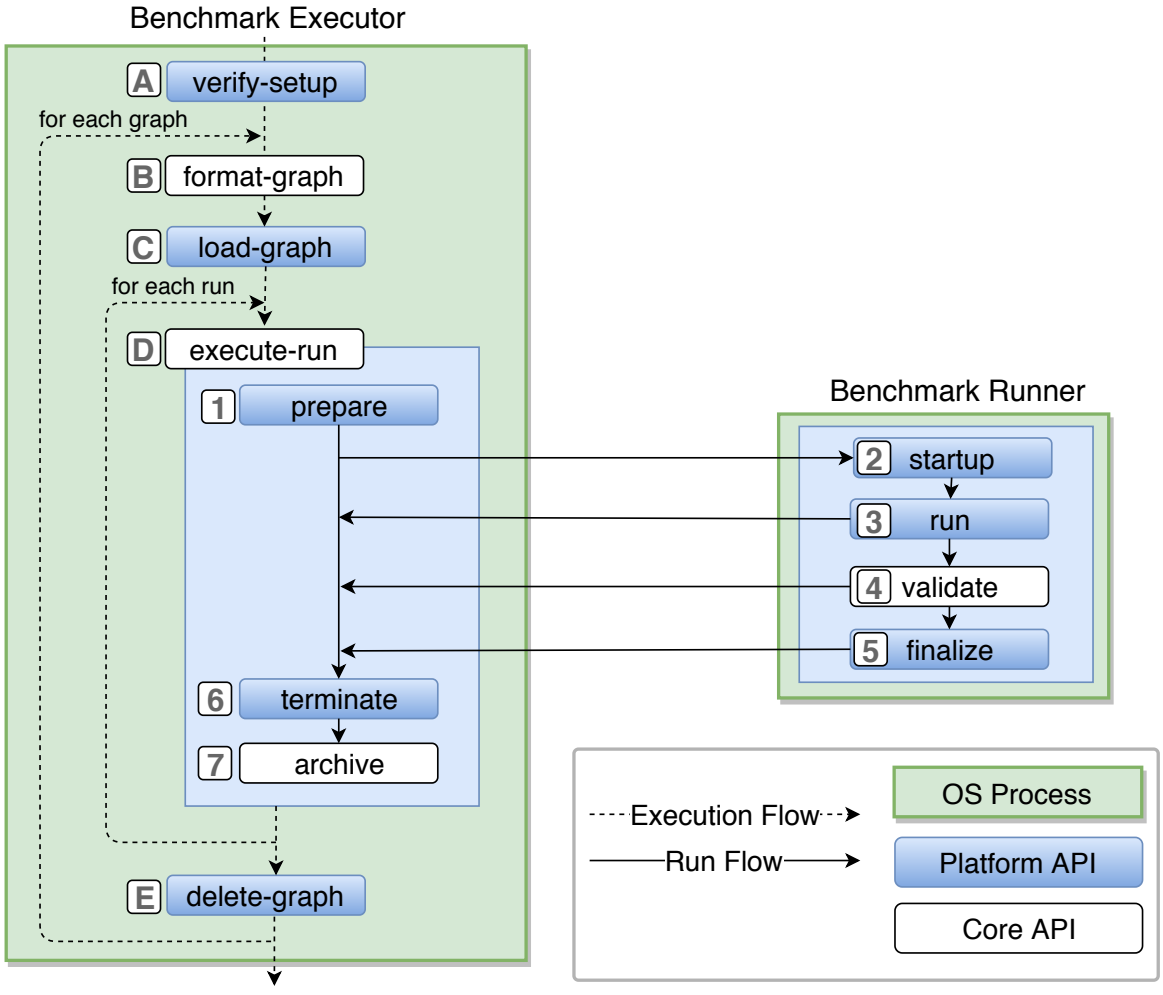


Figure 3.2: Benchmark execution in the Graphalytics benchmark suite.

3.3.1 Execution Flow

After a benchmark is loaded, the benchmark suite analyzes the exact composition of the benchmark. Each benchmark consists of a number of benchmark runs, which will be grouped by the graph dataset used by that benchmark run. For each graph dataset, the input data of that dataset will be loaded only once, and be reused for all corresponding benchmarks runs, before finally being removed.

[A] Verify-setup: The benchmark suite verifies that the platform and the environment are properly set up based on the prerequisites defined in the platform driver.

[B] Format-graph: The benchmark suite minimizes the "input data" into "formatted dataset" (see more in Section 3.3.2) by removing unused vertex and edge properties.

[C] Load-graph: The platform converts the "formatted data" into any platform-specific data format and loads a graph dataset into a storage system, which can be either a local file system, a share file system or a distributed file system. This step corresponds to the "Loading" step of a graph processing job described in Section 2.5.2.

[D] Execute-run: The platform executes a benchmark run with a specific algorithm and dataset (see more details in Section 3.3.2). All benchmark runs using the same input dataset can use the prepared graph dataset during the "load-graph" step.

[E] Delete-graph: The platform unloads a graph dataset from the storage system, as part of the cleaning up process after all benchmark runs on that graph dataset have been completed.

Note that "load-graph" and "delete-graph" are platform-specific API, which can be implemented in the platform driver via the "Platform" interface, whereas "execute-run" is a uniform step for all platforms.

3.3.2 Run Flow

The execution of each benchmark run consists of seven steps in total, i.e., "prepare", "startup", "run", "validate", "finalize", "terminate", and "archive". To ensure the stability, the benchmark suite only prepares for the benchmark, and terminates the benchmark run. Each benchmark run is partially executed in an isolated operating-system process, such that a timed-out job can be terminated properly.

[1] Prepare: The platform requests computation resources from the cluster environment and makes the background applications ready.

[2] Startup: The platform configures the benchmark run with regard to real-time cluster deployment information, e.g., input directory, output directory and log directory.

[3] Run: The platform runs a graph-processing job as defined in the benchmark run. The graph-processing job must complete within the time-out duration, or the benchmark run will fail. This step corresponds to the "Running" step of a graph processing job described in Section 2.5.2.

[4] Validate: The benchmark suite validates the platform output with the validation data. The system under test must succeed in this step, or the benchmark run will fail.

[5] Finalize: The platform reports the benchmark information and makes the environment ready for the next benchmark run.

[6] Terminate: The platform forcibly stops the benchmark job and cleans up the environment, given that the time-out has been reached.

[7] Archive: The benchmark suite archives the benchmark results, gathering information regarding performance metrics and failure indications.

Note that "prepare", "startup", "run", "finalize", "terminate", and "archive" are platform-specific API, which can be implemented in the platform driver via the "Platform" interface, whereas "archive" is a uniform step for all platforms.

3.3.3 Data Flow

The graph datasets go through a series of execution steps during the benchmark execution, and in the process of which the format, the representation, and the content of the graph dataset change accordingly.

For each graph, the input datasets and the validation datasets are publicly available benchmark resources.

- **Input data:** The "input data" consists of a vertex file and edge file in EVLP format, as defined in Section 2.2.2.
- **Validation data:** The "validation data" consists of correct outputs for all six core algorithm, as defined in Section 2.4.

The input dataset can be converted into the following format during the benchmark.

- **Formatted data:** The "input data" can plausibly contain dozens of vertex and edge properties. During the "load-graph" step, the benchmark suite identifies for each algorithms which properties are needed and which are not, and minimizes the "input data" into the "formatted data". The "formatted data" is cached in the storage system for future uses.
- **Loaded data:** The minimized "formatted data" is loaded into a storage system during the "load-graph" step, which can either be a local file system, a share file system or a distributed file system.
- **Output data:** The "output data" is the output of a graph-processing job being benchmarked during the "process" step. The "output data" is compared to the "validation data" to ensure the correctness of the benchmark execution.

3.3.4 Failure Indication

Failures can occur during the benchmark for many reasons. The benchmark suite logs the benchmark execution and classifies the type of failures.

- `DAT`: "Data failure" occurs when the "format-graph" step fails to generate "formatted-graph", or the "load-graph" step fails to complete correctly. For example, "input graph" can be missing or simply be misplaced, or alternatively the conversion from "input-graph" to "formatted-graph" could be prematurely interrupted, which leads to data corruption.
- `INI`: "Initialization failure" occurs when the platform fails to properly make the environment ready for the benchmark during the "prepare" or "startup" step. For example, the deployment system may fail to allocate the cluster resources needed.
- `EXE`: "Execution failure" occurs when the execution of the benchmark run fails to complete during the "run" step.
- `TIM`: "Time-out failure" occurs when the pre-defined time-out duration is reached during the "run" step.
- `COM`: "Completion failure" occurs when output results are incomplete or cannot be found at all. For example, outputs from some compute nodes can be fetched incorrectly.
- `VAL`: "Validation failure" occurs when the "output data" is returned by the system, but fails the validation during the "validate" step.
- `MET`: "Metric failure" occurs when the compulsory performance metrics are missing during the "archive" step. For example, the log files containing the information can be non-existing or corrupted.

3.4 Benchmark Result

A complete result for the Graphalytics benchmark includes at least the following information:

1. Target scale (T).
2. Environment specification, including number and type of CPUs, amount of memory, type of network, etc.
3. Versions of the platform and Graphalytics drivers used in the experiments.
4. Any non-default configuration options for the platform required to reproduce the system under test.
5. For every benchmark job:
   (a) Job specification, i.e., dataset and algorithm.
   (b) For every platform run, report the measured processing time, makespan, and whether the run breached the Graphalytics SLA.
   (c) (optional) `Granula` archives for each platform run, enabling deep inspection, visualization, modeling, and sharing of performance data.

Future versions of the benchmark specification will include a Full Disclosure Report template and a process for submitting official Graphalytics results. A sample data format can be found in Appendix B.

In this chapter, we describe how to participate in Graphalytics-based competitions. For more details, check out the competition specification document [38].

Graphalytics defines several *official competitions* (Section 4.1), which are open globally to everyone who satisfies the rules for participation. Each competition ranks the benchmark submission based on a competition method defined in Section 4.2.

4.1 Official Graphalytics Competitions

Currently, Graphalytics defines two official competitions: (1) the Global LDBC Competition and (2) the Global Graphalytics Competition.

4.1.1 The Global LDBC Competition

The Global LDBC Competition is maintained by LDBC, in particular by the Graphalytics Task Force. By the rules of the LDBC charter [33], the competition method follows the single value-of-merit approach described in Section 4.2.1, and focuses on two primary metrics: **"performance"** and **"cost-performance"**.

The competition reports the following list of metrics:

1. (informative only) Full disclosure of the "system under test" (platform + environment).
2. (informative only) *Target scale* of the benchmark.
3. (informative only) *Date* of the benchmark execution.
4. (flagship value-of-merit) *Performance metric*, as summarized from "EVPS" of all benchmarked jobs.
5. (capability value-of-merit) *Cost-performance metric*, as summarized from "PPP" of all benchmarked jobs.
6. (informative only) Three *performance metrics*, as summarized from T_l, T_m, and T_p respectively.

Maintained by: LDBC, `ldbcouncil.org`.

Audience: The LDBC Competition accepts submissions from a global audience.

4.1.2 The Global Graphalytics Competition

The Global Graphalytics Competition is maintained by the Graphalyticsteam. The competition method follows the tournament-based approach described in Section 4.2.2, and focuses on two primary scores: **"performance"** and **"cost-performance"**.

The Graphalytics consists of a number of *matches*, where each match represents a type of experiment that focuses on a specific performance characteristic that is common across all systems, for example, the EVPS of the BFS algorithm on a Datagen dataset. Each match consists of a set of instances, with the **tournament score** being for each system the sum of **instance scores** accumulated by the platform across all matches in which it participates. Each *instance* is a head-to-head comparison between two systems, for example, comparing the EVPS of any algorithm-dataset for the pair (Giraph, GraphX): the winner receives 1 point, the loser 0 points, and a draw rewards each platform with 0.5 points each.

1. (informative only) Full disclosure of the "system under test" (platform + environment).
2. (informative only) *Target scale* of the benchmark.
3. (informative only) *Date* of the benchmark execution.

4. (ranking) *Performance score*, by comparing pair-wisely "EVPS" of all benchmarked jobs.

5. (ranking) *Cost-performance score*, by comparing pair-wisely "PPP" of all benchmarked jobs.

Maintained by: Graphalytics, `graphalytics.org`.

Audience: The Global Graphalytics Competition accepts submissions from a global audience.

4.2 Competition Method

Different competition methods have been developed to performance comparison in many application domains. Comparing multiple platforms across multiple performance metrics is not trivial. Two major approaches exist for this task: (i) creating a compound metric, typically by weighting the multiple metrics, and comparing multiple platforms using only this single-value-of-merit, and (ii) using a tournament format that allows for multiple participants (platforms) to be compared across multiple criteria.

The former requires metrics to be easy to compare and compose, that is, to be normalized, to be similarly distributed, to have the same meaning of better (e.g., lower values), to be of importance universally recognized across the field of practice so that weights can be easily ascribed. The latter requires a good tournament format, which does not favor any of the participants, and which does not make participation cumbersome through a large set of rules.

4.2.1 Single Value-of-merit Approach

Where metrics (see Section 2.5.3) are collected repeatedly, e.g., each combination of algorithm and dataset, a single value-of-merit can be summarized following the typical processes of benchmarking HPC systems [27]:

- For **Performance metrics,** the *arithmetic mean* across all data.
- For **Throughput metrics,** because they are rate metrics, in two consecutive steps:
  1. let a be the *arithmetic mean* of the performance metric (e.g., processing time) and w be the (constant, total) workload (e.g., count of edges plus vertices),
  2. report the *ratio* between w and a as the throughput metric.

  In other words, instead of averaging the rate per sample, that is, $EVPS_i$ for sample i, Graphalytics first averages the performance metric and then reports the rate.
- For **Cost metrics,** the *harmonic mean* across all data. This is because the denominator (e.g., EVPS for PPP) gives meaning to the ratio (TCO is constant across experiments with the same System Under Test), which indicates that the arithmetic mean would be misleading [27, S.3.1.1].
- For **Ratio metrics** such as Speedup, the *geometric mean* across all data.

4.2.2 Tournament-based Approach

In a tournament-based approach, the system performance is ranked by means of competitive tournaments [49]. Generally, a Round-Robin pair-wise tournament [10] (from hereon, *tournament*) of p participants involves a balanced set of (pair-wise) comparisons between the results of each pair of participants; if there are c criteria to compare the participants, there will be $\frac{1}{2} \times c \times p(p-1)$ pair-wise comparisons. In a pair-wise comparison, a pre-defined amount of points (often, 1 or 3) is given to the better (*winner*) participant from the pair. It is also common to give zero points to the worse (*loser*) participant from the pair, and to split the points between participants with equal performance. Similar tournaments have been used for decades in chess competitions, in professional sports leagues such as (European and American) football, etc.

We do not consider here other pair-wise tournaments, such as replicated tournaments [10] and unbalanced comparisons [11], which have been used especially in settings where comparisons are made by human referees and are typically discretized on 5-point Likert scales, and thus are quantitatively less accurate than the Graphalytics measurements.

5 Implementation Instructions

Graphalytics provides a set of benchmark software and resources which are open-source and publicly available. This chapter explains how to work with Graphalytics software and enumerates the available benchmark resources which are necessary for the benchmark.

5.1 Graphalytics Software and Documentation

The Graphalytics team develops and maintains the core Graphalytics software, which facilitates the benchmark process and is extendable for benchmarking and analyzing the performance of various graph processing platforms.

Graphalytics Core The Graphalytics Core contains the core implementation for the Graphalytics benchmark, provides a programmable interface for platform drivers.
Link: `https://github.com/ldbc/ldbc_graphalytics`

Graphalytics Specification The source code for generating this specification.
Link: `https://github.com/ldbc/ldbc_graphalytics_docs`

Reference Implementations We provide two reference implementations.

- SuiteSparse:GraphBLAS [12]: `https://github.com/ldbc/ldbc_graphalytics_platforms_graphblas`
- Umbra [40]: `https://github.com/ldbc/ldbc_graphalytics_platforms_umbra`

Bibliography

[1] Günes Aluç et al. "Diversified Stress Testing of RDF Data Management Systems". In: *ISWC*. 2014, pp. 197–212. DOI: 10.1007/978-3-319-11964-9_13.

[2] Khaled Ammar and M. Tamer Özsu. "WGB: Towards a Universal Graph Benchmark". In: *WBDB*. 2013, pp. 58–72. DOI: 10.1007/978-3-319-10596-3_6.

[3] Timothy Armstrong et al. "LinkBench: A database benchmark based on the Facebook social graph". In: *SIGMOD*. 2013, pp. 1185–1196. DOI: 10.1145/2463676.2465296.

[4] David A. Bader and Kamesh Madduri. "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors". In: *HiPC*. Vol. 3769. Lecture Notes in Computer Science. Springer, 2005, pp. 465–476. DOI: 10.1007/11602569_48.

[5] Scott Beamer, Krste Asanovic, and David A. Patterson. "The GAP Benchmark Suite". In: *CoRR* abs/1508.03619 (2015). arXiv: 1508.03619.

[6] Christian Bizer and Andreas Schultz. "The Berlin SPARQL Benchmark". In: *Int. J. Semantic Web Inf. Syst.* 5.2 (2009), pp. 1–24. DOI: 10.4018/jswis.2009040101.

[7] Mihai Capota et al. "Graphalytics: A Big Data Benchmark for Graph-Processing Platforms". In: *GRADES at SIGMOD*. ACM, 2015, 7:1–7:6. DOI: 10.1145/2764947.2764954.

[8] Meeyoung Cha et al. "Measuring User Influence in Twitter: The Million Follower Fallacy". In: *ICWSM*. The AAAI Press, 2010. URL: http://www.aaai.org/ocs/index.php/ICWSM/ICWSM10/paper/view/1538.

[9] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. "R-MAT: A Recursive Model for Graph Mining". In: *SIAM International Conference on Data Mining (SDM)*. 2004, pp. 442–446. DOI: 10.1137/1.9781611972740.43.

[10] H. A. David. "The Method of Paired Comparisons". In: *Fifth Conference on the Design of Experiments in Army Research Development and Testing*. US Army Office of Ordnance Research. 1960, pp. 1–17.

[11] Herbert A. David. "Ranking from unbalanced paired-comparison data". In: *Biometrika* 74 (1987), pp. 432–436.

[12] Timothy A. Davis. "Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra". In: *ACM Trans. Math. Softw.* 45.4 (2019), 44:1–44:25. DOI: 10.1145/3322125. URL: https://doi.org/10.1145/3322125.

[13] Miyuru Dayarathna and Toyotaro Suzumura. "Graph database benchmarking on cloud environments with XGDBench". In: *Autom. Softw. Eng.* 21.4 (2014), pp. 509–533. DOI: 10.1007/s10515-013-0138-7.

[14] Nadav Eiron, Kevin S. McCurley, and John A. Tomlin. "Ranking the web frontier". In: *WWW*. ACM, 2004, pp. 309–318. DOI: 10.1145/988672.988714.

[15] Assaf Eisenman et al. "Parallel Graph Processing: Prejudice and State of the Art". In: *ICDE*. ACM, 2016, pp. 85–90. DOI: 10.1145/2851553.2851572.

[16] Benedikt Elser and Alberto Montresor. "An evaluation study of BigData frameworks for graph processing". In: *Big Data*. 2013, pp. 60–67. DOI: 10.1109/BigData.2013.6691555.

[17] Orri Erling et al. "The LDBC Social Network Benchmark: Interactive Workload". In: *SIGMOD*. 2015, pp. 619–630. DOI: 10.1145/2723372.2742786.

[18] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. "The Case Against Specialized Graph Analytics Engines". In: *CIDR*. 2015. URL: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper20.pdf.

[19] Michael Ferdman et al. "Clearing the clouds: A study of emerging scaleout workloads on modern hardware". In: *ASPLOS*. 2012, pp. 37–48. DOI: 10.1145/2150976.2150982.

[20] Ahmad Ghazal et al. "BigBench: Towards an industry standard benchmark for Big Data analytics". In: *SIGMOD*. 2013, pp. 1197–1208. DOI: 10.1145/2463676.2463712.

[21] Joseph E. Gonzalez et al. “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”. In: *OSDI*. USENIX Association, 2012, pp. 17–30. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez.

[22] Yong Guo and Alexandru Iosup. “The Game Trace Archive”. In: *NETGAMES*. IEEE, 2012, pp. 1–6. DOI: 10.1109/NetGames.2012.6404027.

[23] Yong Guo et al. “An Empirical Performance Evaluation of GPU-Enabled Graph-Processing Systems”. In: *CCGrid*. 2015, pp. 423–432. DOI: 10.1109/CCGrid.2015.20.

[24] Yong Guo et al. “How Well Do Graph-Processing Platforms Perform?” In: *IPDPS*. 2014, pp. 395–404. DOI: 10.1109/IPDPS.2014.49.

[25] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. “LUBM: A benchmark for OWL knowledge base systems”. In: *J. Web Semant.* 3.2-3 (2005), pp. 158–182. DOI: 10.1016/j.websem.2005.06.005.

[26] Minyang Han et al. “An Experimental Comparison of Pregel-like Graph Processing Systems”. In: *PVLDB* 7.12 (2014), pp. 1047–1058.

[27] Torsten Hoefler and Roberto Belli. “Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results”. In: *SC*. 2015, 73:1–73:12. DOI: 10.1145/2807591.2807644.

[28] Sungpack Hong et al. “PGX.D: A fast distributed graph processing engine”. In: *SC*. 2015, 58:1–58:12. DOI: 10.1145/2807591.2807620.

[29] Alexandru Iosup et al. *LDBC Graphalytics graphs*. https://hdl.handle.net/11112/7ec6a51e-6fdb-bf8d-4507-456ccadc9291. DOI: 10.25606/SURF.e8e60a7e282917f5.

[30] Alexandru Iosup et al. “LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms”. In: *PVLDB* 9.13 (2016), pp. 1–12.

[31] Alexandru Iosup et al. “The LDBC Graphalytics Benchmark”. In: *CoRR* abs/2011.15028 (2020). arXiv: 2011.15028. URL: https://arxiv.org/abs/2011.15028.

[32] Alekh Jindal et al. “VERTEXICA: Your Relational Friend for Graph Analytics!” In: *PVLDB* 7.13 (2014), pp. 1669–1672. DOI: 10.14778/2733004.2733057.

[33] LDBC. “Byelaws of the Linked Data Benchmark Council v1.1”. In: (2017). http://ldbcouncil.org/sites/default/files/ldbc-byelaws-1.1.pdf.

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

[35] Yi Lu et al. “Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation”. In: *PVLDB* 8.3 (2014), pp. 281–292. URL: http://www.vldb.org/pvldb/vol8/p281-lu.pdf.

[36] Zijian Ming et al. “BDGS: A Scalable Big Data Generator Suite in Big Data Benchmarking”. In: *WBDB*. 2013, pp. 138–154. DOI: 10.1007/978-3-319-10596-3_11.

[37] Richard C. Murphy et al. “Introducing the Graph 500”. In: *Cray Users Group (CUG)* 19 (2010), pp. 45–74. URL: https://cug.org/5-publications/proceedings_attendee_lists/CUG10CD/pages/1-program/final_program/CUG10_Proceedings/pages/authors/11-15Wednesday/14C-Murphy-paper.pdf.

[38] Ahmed Musaafir et al. *Specification of different Graphalytics Competitions*. Presented at ICT.OPEN in Amersvoort. 2018. URL: https://graphalytics.org/assets/spec-graphalytics-competitions.pdf.

[39] Lifeng Nai et al. “GraphBIG: Understanding graph computing in the context of industrial solutions”. In: *SC*. 2015, 69:1–69:12. DOI: 10.1145/2807591.2807626.

[40] Thomas Neumann and Michael J. Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *Umbra*. 2020. URL: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf.

[41] Lawrence Page et al. “The PageRank citation ranking: Bringing order to the Web.” In: (1999).

[42] Tilmann Rabl et al. “The Vision of BigBench 2.0”. In: *DanaC*. 2015, 3:1–3:4. DOI: 10.1145/2799562.2799642.

[43] Usha Raghavan, Réka Albert, and Soundar Kumara. “Near linear time algorithm to detect community structures in large-scale networks”. In: *Physical Review E* 76.3 (2007), p. 036106.

[44] Sherif Sakr et al. *Large-Scale Graph Processing Using Apache Giraph*. Springer, 2016. ISBN: 978-3-319-47430-4. DOI: 10.1007/978-3-319-47431-1.

[45] Nadathur Satish et al. “Navigating the maze of graph analytics frameworks using massive datasets”. In: *SIGMOD*. 2014, pp. 979–990. DOI: 10.1145/2588555.2610518.

[46] Michael Schmidt et al. “SP2Bench: A SPARQL Performance Benchmark”. In: *ICDE*. 2009, pp. 222–233. DOI: 10.1109/ICDE.2009.28.

[47] Narayanan Sundaram et al. “GraphMat: High performance graph analytics made productive”. In: *PVLDB* 8.11 (2015), pp. 1214–1225. DOI: 10.14778/2809974.2809983.

[48] Gábor Szárnyas et al. “An early look at the LDBC Social Network Benchmark’s Business Intelligence workload”. In: *GRADES-NDA at SIGMOD*. ACM, 2018, 9:1–9:11. DOI: 10.1145/3210259.3210268.

[49] Louis Leon Thurstone. “Psychophysical Analysis”. In: *American Journal of Psychology* 38 (1927), pp. 368–89.

[50] *TPC Pricing-Standard Specification*. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-pricing_v2.0.0.pdf.

[51] Lei Wang et al. “BigDataBench: A Big Data benchmark suite from internet services”. In: *HPCA*. 2014, pp. 488–499. DOI: 10.1109/HPCA.2014.6835958.

[52] Reynold S. Xin et al. “GraphX: A resilient distributed graph system on Spark”. In: *GRADES at SIGMOD*. 2013. DOI: 10.1145/2484425.2484427.

A Pseudo-code for Algorithms

This chapter contains pseudo-code for the algorithms described in Section 2.3. In the following sections, a graph G consists of a set of vertices V and a set of edges E. For undirected graphs, each edge is bidirectional, so if $(u, v) \in E$ then $(v, u) \in E$. Each vertex has a set of outgoing neighbors $N_{\text{out}}(v) = \{u \in V | (v, u) \in E\}$ and a set of incoming neighbors $N_{\text{in}}(v) = \{u \in V | (u, v) \in E\}$.

A.1 Breadth-First Search (BFS)

```
   input: graph G = (V, E), vertex root
   output: array depth storing vertex depths
 1: for all v ∈ V do
 2:     depth[v] ← ∞
 3: end for
 4: Q ← CREATE_QUEUE()
 5: Q.PUSH(root)
 6: depth[root] ← 0
 7: while Q.SIZE > 0 do
 8:     v ← Q.POP_FRONT( )
 9:     for all u ∈ N_out(v) do
10:         if depth[u] = ∞ then
11:             depth[u] ← depth[v] + 1
12:             Q.PUSH_BACK(u)
13:         end if
14:     end for
15: end while
```

A.2 PageRank (PR)

```
   input: graph G = (V, E), integer max_iterations
   output: array rank storing PageRank values
 1: for all v ∈ V do
 2:     rank[v] ← 1/|V|
 3: end for
 4: for i = 1, ..., max_iterations do
 5:     sink_sum ← 0
 6:     for all w ∈ V do
 7:         if |N_out(w)| = 0 then
 8:             sink_sum ← sink_sum + rank[w]
 9:         end if
10:     end for
11:     for all v ∈ V do
12:         new_rank[v] ← (1−d)/|V| + d · Σ_{u∈N_in(v)} rank[u]/|N_out(u)| + d/|V| · sink_sum
13:     end for
14:     rank ← new_rank
15: end for
```

A.3 Weakly Connected Components (WCC)

input: graph $G = (V, E)$
output: array *comp* storing component labels

for all $v \in V$ **do**
  $comp[v] \leftarrow v$
end for
repeat
  $converged \leftarrow$ true
  for all $v \in V$ **do**
    for all $u \in N_{\text{in}}(v) \cup N_{\text{out}}(v)$ **do**
      if $comp[v] > comp[u]$ **then**
        $comp[v] \leftarrow comp[u]$
        $converged \leftarrow$ false
      end if
    end for
  end for
until *converged*

A.4 Local Clustering Coefficient (LCC)

input: graph $G = (V, E)$
output: array *lcc* storing LCC values

for all $v \in V$ **do**
  $d \leftarrow |N_{\text{in}}(v) \cup N_{\text{out}}(v)|$
  if $d \geq 2$ **then**
    $t \leftarrow 0$
    for all $u \in N_{\text{in}}(v) \cup N_{\text{out}}(v)$ **do**
      for all $w \in N_{\text{in}}(v) \cup N_{\text{out}}(v)$ **do**
        if $(u, w) \in E$ **then** ▷ Check if edge (u, w) exists
          $t \leftarrow t + 1$ ▷ Found triangle $v - u - w$
        end if
      end for
    end for
    $lcc[v] \leftarrow \frac{t}{d(d-1)}$
  else
    $lcc[v] \leftarrow 0$ ▷ No triangles possible
  end if
end for

A.5 Community Detection using Label Propagation (CDLP)

input: graph $G = (V, E)$, integer *max_iterations*
output: array *labels* storing vertex communities

for all $v \in V$ **do**
  labels$[v] \leftarrow v$
end for
for $i = 1, \ldots,$ *max_iterations* **do**
  for all $v \in V$ **do**
    $C \leftarrow$ CREATE_HISTOGRAM()
    for all $u \in N_{\text{in}}(v)$ **do**
      C.ADD(*labels*$[u]$)
    end for
    for all $u \in N_{\text{out}}(v)$ **do**
      C.ADD(*labels*$[u]$)
    end for
    freq $\leftarrow C$.GET_MAXIMUM_FREQUENCY( ) ▷ Find maximum frequency of labels
    candidates $\leftarrow C$.GET_LABELS_FOR_FREQUENCY(*freq*) ▷ Find labels with max. frequency
    new_labels$[v] \leftarrow$ MIN(*candidates*) ▷ Select smallest label
  end for
  labels $\leftarrow$ *new_labels*
end for

A.6 Single-Source Shortest Paths (SSSP)

input: graph $G = (V, E)$, vertex *root*, edge weights *weight*
output: array *dist* storing distances

for all $v \in V$ **do**
  dist$[v] \leftarrow \infty$
end for
$H \leftarrow$ CREATE_HEAP()
H.INSERT(root, 0)
dist[*root*] $\leftarrow 0$
while H.SIZE > 0 **do**
  $v \leftarrow H$.DELETE_MINIMUM( ) ▷ Find vertex v in H such that *dist*$[v]$ is minimal
  for all $w \in N_{\text{out}}(v)$ **do**
    if *dist*$[w] >$ *dist*$[v] +$ *weight*$[v, w]$ **then**
      dist$[w] \leftarrow$ *dist*$[v] +$ *weight*$[v, w]$
      H.INSERT(w, *dist*$[w]$)
    end if
  end for
end while

B Data format for Benchmark Results

This appendix shows an example of Graphalytics benchmark results of the reference implementation. Graphalytics benchmark defines a specific data format for the benchmark results. The result is formatted in JSON, and consists of three main components: system under test, benchmark configuration, and experimental results. Listing B.1 depicts the top-level structure of the result format.

Listing B.1: Result Format: Overview

```
{
  "id": "b2940223",
  "system": {...},
  "configuration": {...},
  "result": {...}
}
```

For the system under test, Graphalytics reports the detailed descriptions of the graph analytic platform, the cluster environment, and the benchmark tool.

Listing B.2: Result Format: System Under Test

```
"system": {
  "platform": {
    "name": "reference",
    "acronym": "ref",
    "version": "1.4.0",
    "link": "https://github.com/ldbc/ldbc_graphalytics_platforms_reference"
  },
  "environment": {
    "name": "DAS Supercomputer",
    "acronym": "das5",
    "version": "5",
    "link": "http://www.cs.vu.nl/das5/",
    "machines": [
      {
        "quantity": 20,
        "operating-system": "Centos",
        "cpu": {
          "name": "XEON",
          "cores": "16"
        },
        "memory": {
          "name": "?",
          "size": "40GB"
        },
        "network": {
          "name": "Infiniband",
          "throughput": "10GB/s"
        },
        "storage": {
          "name": "SSD",
          "volume": "20TB"
        },
        "accel": {}
      }
    ]
  },
  "benchmark": {
    "graphalytics-core": {
      "name": "graphalytics-core",
      "version": "1.1.0",
      "link": "https://github.com/ldbc/ldbc_graphalytics"
    },
```

```
    "graphalytics-platforms-reference": {
      "name": "graphalytics-platforms-reference",
      "version": "2.1.0",
      "link": "https://github.com/ldbc/ldbc_graphalytics_platforms_reference"
    }
  }
}
```

For the benchmark configuration, the target scale and the computation resource usage is reported. For each resource type, the baseline resource usage, and the scalablity of that resource type is reported.

Listing B.3: Result Format: Benchmark Configuration

```
"configuration": {
  "target-scale": "L",
  "resources": {
    "cpu-instance": { "name": "cpu-instance", "baseline": 1,  "scalability": true  },
    "cpu-core":     { "name": "cpu-core",     "baseline": 32, "scalability": false },
    "memory":       { "name": "memory",       "baseline": 64, "scalability": true  },
    "network":      { "name": "network",      "baseline": 10, "scalability": false }
  }
}
```

Listing B.4: Result Format: Experiment Result

```
"result": {
  "experiments": {
    "e34252": {
      "id": "e34252",
      "type": "baseline-alg-bfs",
      "jobs": [ "j34252", "j75352", "j23552" ]
    },
    "e75352": {
      "id": "e75352",
      "type": "baseline-alg-pr",
      "jobs": [ "j34252", "j75352", "j23552" ]
    },
    "e23552": {
      "id": "e23552",
      "type": "baseline-alg-cdlp",
      "jobs": [ "j34252", "j75352", "j23552" ]
    }
  },
  "jobs": {
    "j34252": {
      "id": "j34252",
      "algorithm": "bfs",
      "dataset": "D100",
      "scale": 1,
      "repetition": 3,
      "runs": [ "r649352", "r124252", "r124252" ]
    },
    "j75352": {
      "id": "j75352",
      "algorithm": "pr",
      "dataset": "D1000",
      "scale": 1,
      "repetition": 3,
      "runs": [ "r649352", "r124252", "r124252" ]
    },
    "j23552": {
      "id": "j23552",
      "algorithm": "cdlp",
```

```
      "dataset": "G25",
      "scale": 1,
      "repetition": 3,
      "runs": [ "r649352", "r124252", "r124252" ]
    }
  },
  "runs": {
    "r649352": {
      "id": "r649352",
      "timestamp": 1463310828849,
      "success": true,
      "makespan": 23423422,
      "processing-time": 2234
    },
    "r124252": {
      "id": "r124252",
      "timestamp": 1463310324849,
      "success": true,
      "makespan": 2343422,
      "processing-time": 234
    },
    "r643252": {
      "id": "r124252",
      "timestamp": 1463310324849,
      "success": true,
      "makespan": 2343422,
      "processing-time": 234
    }
  }
}
```


For the experimental results, the set of experiments, the underlying jobs, and the corresponding runs are reported.

Table C.1: Overview of related work. (Acronyms: *Reference type*: **S**, study; **B**, benchmark. *Target system, structure*: **D**, distributed system; **P**, parallel system; **MC**, single-node multi-core system; **GPU**, using GPUs. *Input*: **0**, no parameters; **S**, parameters define scale; **E**, parameters define edge properties; **+**, parameters define other graph properties, e.g., clustering coefficient. *Datasets/Algorithms*: **Rnd**, reason for selection not explained; **Exp**, selection guided by expertise; **1-stage**, data-driven selection; **2-stage**, 2-stage data- and expertise-driven process. *Scalability tests*: **W**, weak; **S**, strong; **V**, vertical; **H**, horizontal.)

Reference (chronological order)		Target System (R1)		Design (R2)				Tests (R3)		(R4)
	Name [Publication]	Structure	Programming	Input	Datasets	Algo.	Scalable?	Scalability	Robustness	Renewal
B	CloudSuite [19], only graph elements	D/MC	PowerGraph	S	Rnd	Exp	—	No	No	No
S	Montresor et al. [16]	D/MC	3 classes	0	Rnd	Exp	—	No	No	No
B	HPC-SGAB [4]	P	—	S	Exp	Exp	—	No	No	No
B	Graph500	P/MC/GPU	—	S	Exp	Exp	—	No	No	No
B	GreenGraph500	P/MC/GPU	—	S	Exp	Exp	—	No	No	No
B	WGB [2]	D	—	SE+	Exp	Exp	1B Edges	No	No	No
S	**Own prior work** [24, 23, 7]	**D/MC/GPU**	**10 classes**	**S**	**Exp**	**1-stage**	**1B Edges**	**W/S/V/H**	**No**	**No**
S	Özsu et al. [26]	D	Pregel	0	Exp,Rnd	Exp	—	W/S/V/H	No	No
B	BigDataBench [36, 51], only graph elements	D/MC	Hadoop	S	Rnd	Rnd	—	S	No	No
S	Satish et al. [45]	D/MC	6 classes	S	Exp,Rnd	Exp	—	W	No	No
S	Lu et al. [35]	D	4 classes	S	Exp,Rnd	Exp	—	S	No	No
B	GraphBIG [39]	P/MC/GPU	System G	S	Exp	Exp	—	No	No	No
S	Cherkasova et al. [15]	MC	Galois	0	Rnd	Exp	—	No	No	No
B	**LDBC Graphalytics** (this work)	**D/MC/GPU**	**10+ classes**	**SE+**	**2-stage**	**2-stage**	**Process**	**W/S/V/H**	**Yes**	**Yes**

Table C.1, which is reproduced from [30], summarizes and compares Graphalytics with previous studies and benchmarks for graph analysis systems. R1–R5 are the requirements formulated in Section 2.1. As the table indicates, there is no alternative to Graphalytics in covering requirements R1–R4. We also could not find evidence of requirement R5 being covered by other systems than LDBC. While there have been a few related *benchmark proposals* (marked "B"), these either do not *focus* on graph analysis, or are much narrower in scope (e.g., only BFS for Graph500). There have been comparable *studies* (marked "S") but these have not attempted to define—let alone maintain—a benchmark, its specification, software, testing tools and practices, or results. Graphalytics is not only industry-backed but also has industrial strength, through its detailed execution process, its metrics that characterize robustness in addition to scalability, and a renewal process that promises longevity. Graphalytics is being proposed to SPEC as well, and BigBench [20, 42] explicitly refers to Graphalytics as its option for future benchmarking of graph analysis platforms.

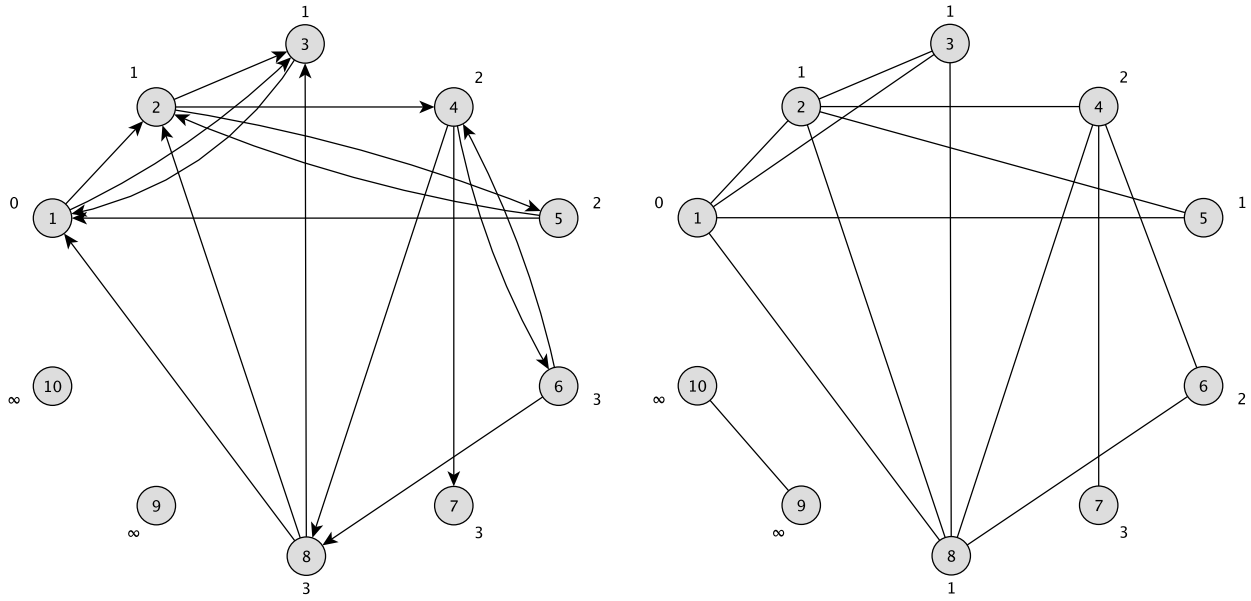
Previous studies typically tested the open-source platforms Giraph [44], GraphX [52], and PowerGraph [21], but our contribution here is that vendors (Oracle, Intel, IBM) in our evaluation have themselves tuned and tested their implementations for PGX [28], GraphMat [47] and OpenG [39]. We are aware that the database community has started to realize that with some enhancements, RDBMS technology could also be a contender in this area [18, 32], and we hope that such systems will soon get tested with Graphalytics.

Graphalytics complements the many existing efforts focusing on graph databases, such as LinkBench [3], XGDBench [13], and LDBC SNB [17, 48]; efforts focusing on RDF graph processing, such as LUBM [25], the Berlin SPARQL Benchmark [6], SP^{2}Bench [46], and WatDiv [1] (targeting also graph databases); and community efforts such as the TPC benchmarks. Whereas all these prior efforts are interactive database query benchmarks, Graphalytics focuses on algorithmic graph analysis and on different platforms which are not necessarily database systems, whose distributed and highly parallel aspects lead to different design trade-offs.

The GAP Benchmark Suite [5] targets six graph kernels: BFS, SSSP, PR, WCC, triangle count, and betweenness centrality. The first four are present in Graphalytics, while the *triangle count* kernel shares many challenges with the LCC algorithm. (*Betweenness centrality* was also considered for Graphalytics but it necessitates using approximation methods for large graphs, therefore automated validation of results is not possible.)

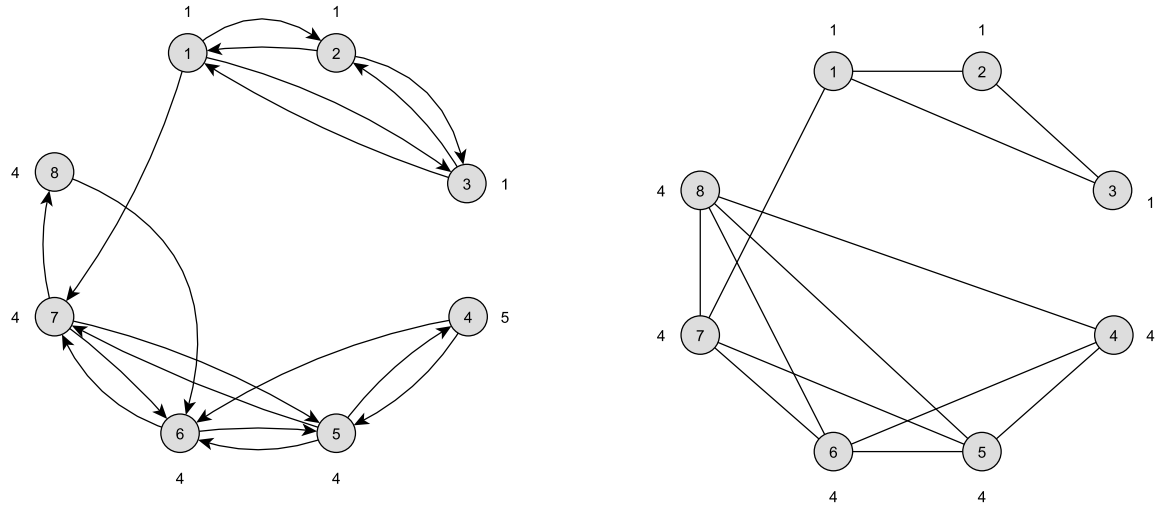
D Example Graphs for Validation

In this chapter, we provide test graphs containing the expected results for BFS (Figure D.1), CDLP (Figure D.2), LCC (Figure D.3), SSSP (Figure D.5), WCC (Figure D.6), and a common example for all six algorithms (Figure D.7). The PageRank test graphs (`test-pr-directed` and `test-pr-undirected`) are provided in the data sets but not displayed here due to their relatively large sizes (50 nodes).



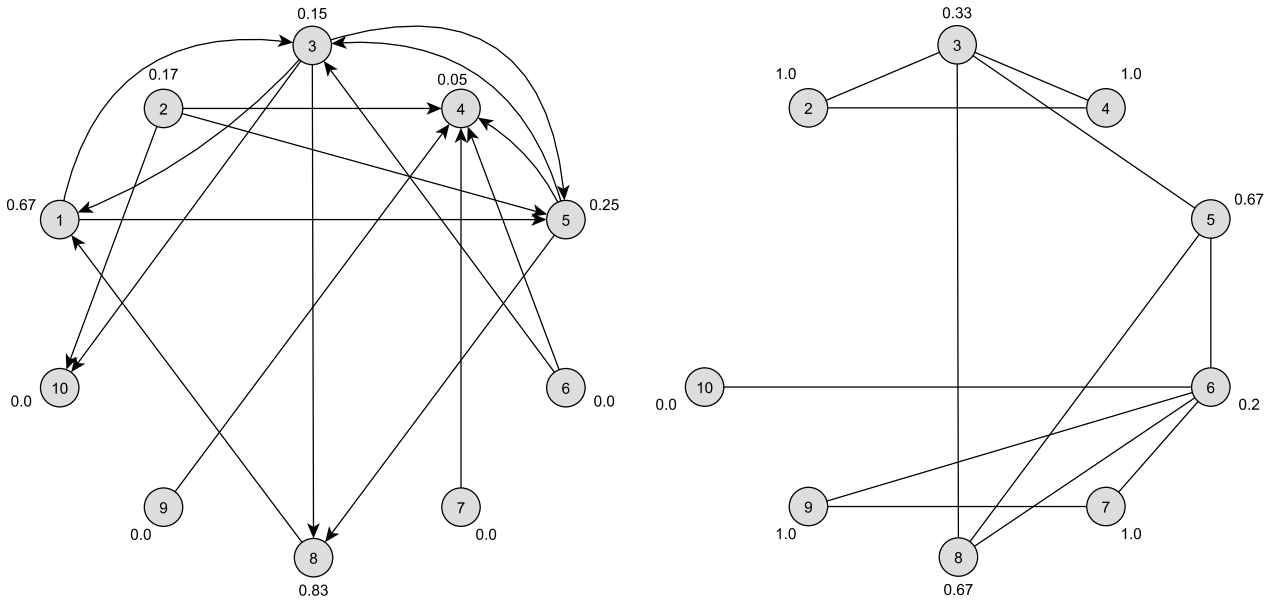
(a) Directed (`text-bfs-directed`) (b) Undirected (`text-bfs-undirected`)

Figure D.1: Test graphs for BFS. Source vertex: 1.



(a) Directed (`text-cdlp-directed`) (b) Undirected (`text-cdlp-undirected`)

Figure D.2: Test graphs for CDLP. Maximum number of iterations: 5.



(a) Directed (`text-lcc-directed`)

(b) Undirected (`text-lcc-undirected`)

Figure D.3: Test graphs for LCC.

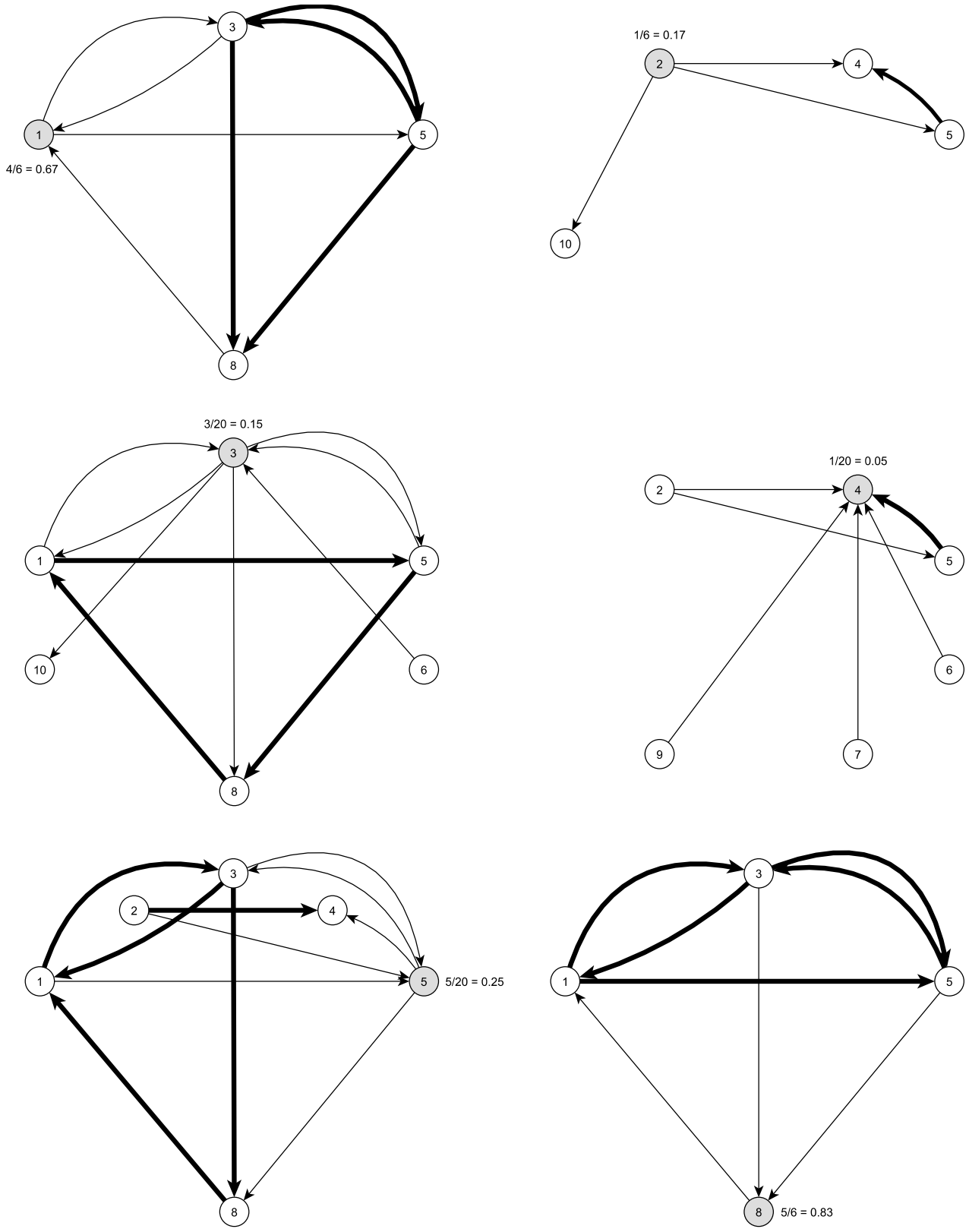
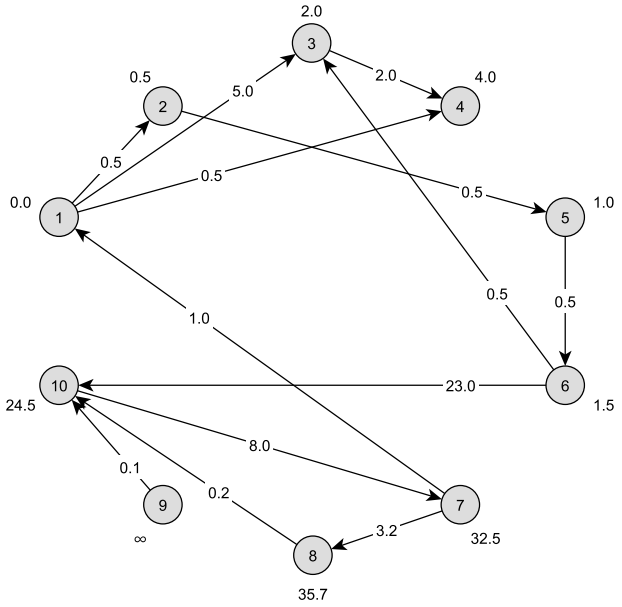
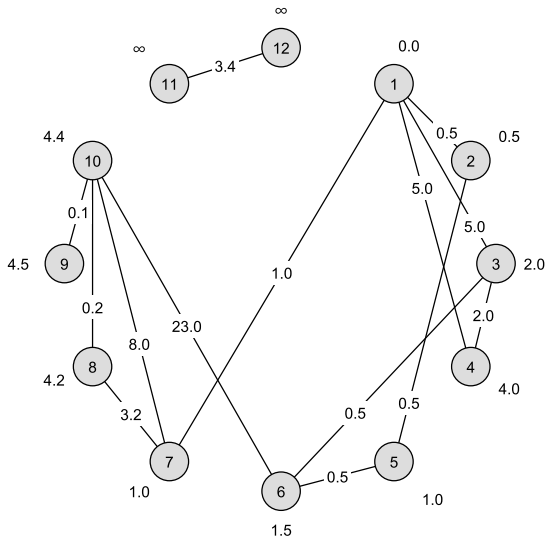


Figure D.4: Detailed example of LCC values on a directed graph. Each graph represents a projected subgraph with a selected vertex (colored in gray) and its neighbors. Thick edges (denote the edges between the neighbors, while thin edges denote the edges from the vertex to its neighbors. As discussed in Section 2.3.5, the set of neighbors is determined without taking directions into account, but each neighbor is only counted once. However, directions are enforced when determining the number of edges (thick) between neighbors. Note that vertices 6, 7, 9, and 10 have an LCC value of 0.00 and are therefore omitted from the visualization.

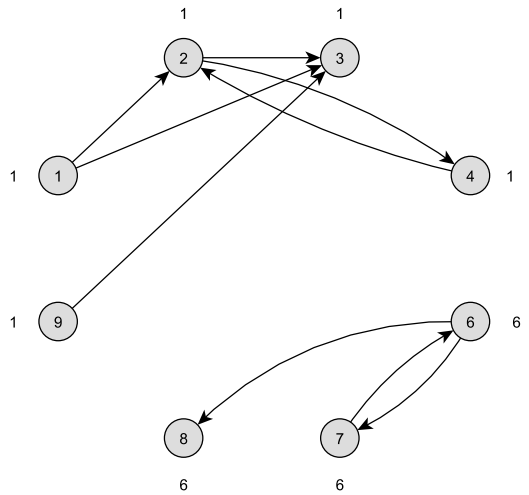


(a) Directed (`text-sssp-directed`)

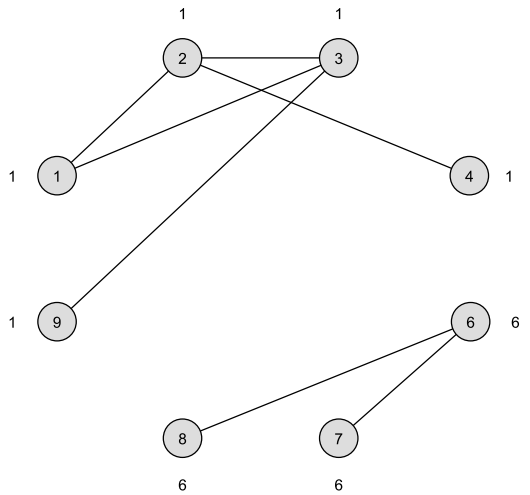


(b) Undirected (`text-sssp-undirected`)

Figure D.5: Test graphs for SSSP. Source vertex: 1.

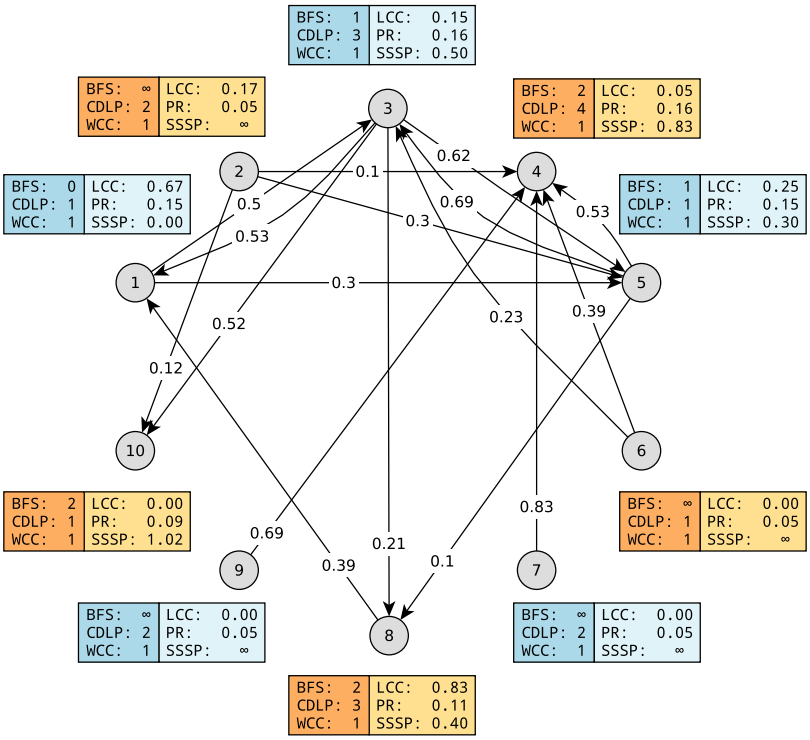


(a) Directed (`text-wcc-directed`). Note that due to the semantics of the WCC algorithm, the direction of the edges is not taken into account, i.e., they are treated as undirected edges.

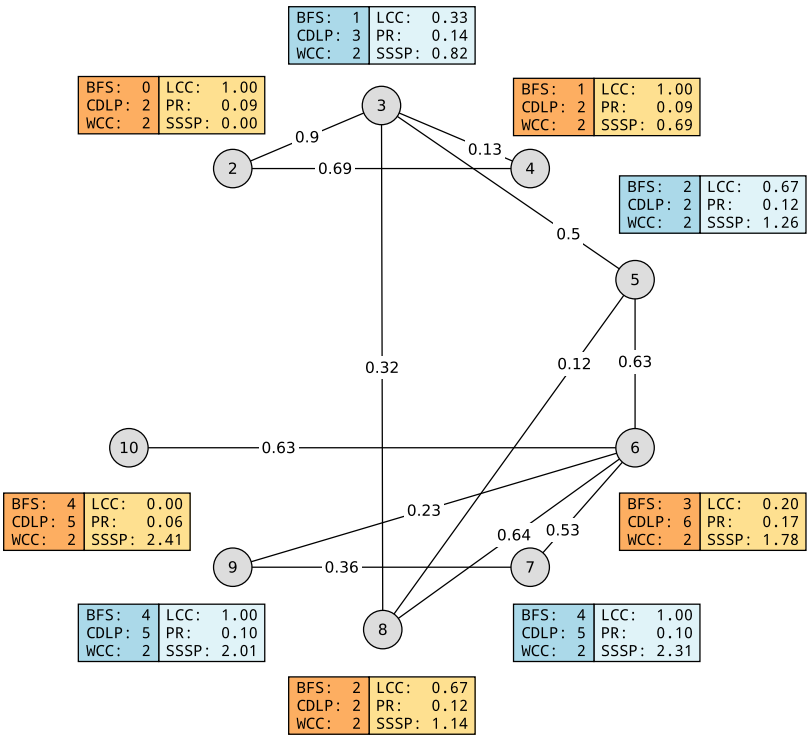


(b) Undirected (`text-wcc-undirected`)

Figure D.6: Test graphs for WCC. Remark: there are 8 nodes indexed between 1 and 9 but number 5 is not assigned to any of the nodes.



(a) Directed (`example-directed`). Algorithm parameters – BFS: source vertex = 1. CDLP: maximum number of iterations = 2. PR: $d = 0.85$, maximum number of iterations = 2. SSSP: source vertex = 1.



(b) Undirected (`example-undirected`). Algorithm parameters – BFS: source vertex = 2. CDLP: maximum number of iterations = 2. PR: $d = 0.85$, maximum number of iterations = 2. SSSP: source vertex = 2.

Figure D.7: Common examples for the six core graph algorithms.