Summary

Metric	Ground Truth	My Compute	Notes on Difference
Nodes	7,115	7,115	Perfect match → dataset parsed correctly.
Edges	103,689	103,689	Perfect match → no edges lost/skipped.
Largest WCC (nodes)	7,066 (0.993)	7,066	Exact match → connected components computed correctly.
Largest WCC (edges)	103,663 (1.000)	103,663	Exact match.
Largest SCC (nodes)	1,300 (0.183)	1,300	Exact match.
Largest SCC (edges)	39,456 (0.381)	39,456	Exact match.
Avg. clustering coefficient	0.1409	0.14089	Perfect match
Number of triangles	608,389	608,389	Perfect match.
Fraction of closed triangles	0.04564	0.04182	Noticeable difference (~9%). Maybe due to formula implementation: some libraries normalize differently (by triplets vs connected triples).
Diameter	7	7	Perfect match
Effective diameter (90%)	3.8	4	Calculated the Effective diameter as the 90th percentile of all pairs of shortest paths, which cannot be a float

Weakly Connected Components (WCC)

Algorithm Design Choices

The WCC algorithm identifies groups of nodes connected when edge direction is ignored.

- **Undirected edges:** Original and reversed edges were combined to make the graph undirected.
- **Initialization:** Each vertex started with its own ID as the component label.
- **Propagation:** Iteratively, each vertex adopted the minimum component ID among its neighbors.
- **Convergence:** The process repeated until no label changes occurred.
- **Statistics:** Node and edge counts per component were computed to find the largest WCC.

Iterative Implementation in Spark

Implemented using **PySpark DataFrames** with repeated joins and aggregations.
Each iteration:

- Joined edges with component labels to propagate IDs.
- Grouped by vertex to keep the minimum label.
- Checked convergence using `.count()` on differences.

While effective, Spark's batch execution model makes iteration costly due to repeated shuffles and job re-submissions.

Performance and Discrepancies

- Scales with graph size but slows with more iterations or large components.

- Convergence detection (`count`) is expensive.
- Performance can improve using caching or GraphFrames.
- Minor discrepancies may arise from partition skew or join order.

Strongly Connected Components (SCC)

Algorithm Design Choices

The SCC algorithm identifies subgraphs where every vertex is reachable from every other vertex via directed paths.

- **Kosaraju's Algorithm:** Implemented a two-pass DFS approach for clarity and correctness.
- **Data Extraction:** Vertices and edges were collected to Python for efficient in-memory traversal.
- **Pass 1 (Reverse DFS):** Computed finishing order on the reversed graph to determine exploration sequence.
- **Pass 2 (Forward DFS):** Used original graph to assign component IDs in reverse finishing order.
- **Component Analysis:** Counted nodes and edges within each SCC to identify the largest one.

Iterative Implementation

Although the computation was performed outside Spark (in Python), Spark DataFrames were used for input/output handling.
The algorithm builds adjacency lists and performs **iterative depth-first searches** using stacks for both passes, avoiding recursion overhead and enabling efficient traversal.

Performance and Discrepancies

- **Performance:** Suitable for moderate graph sizes since all data is collected to the driver. Not ideal for very large graphs due to memory constraints.
- **Accuracy:** Kosaraju guarantees exact SCC detection but depends on complete data collection.
- **Discrepancies:**
    - Driver memory may limit scalability.
    - Minor overhead from data conversion between Spark and Python objects.

Clustering Metrics & Triangles

Algorithm Design Choices

The clustering coefficient measures how tightly nodes are interconnected, based on the number of triangles (closed triplets) in the graph.

- **Undirected Edges:** Directed edges were converted to undirected and duplicates removed to ensure accurate triangle counting.
- **Adjacency Lists:** Built for efficient neighbor intersection checks.
- **Triangle Detection:** For each edge (u,v)(u, v)(u,v), common neighbors were found using set intersection of adjacency lists, representing triangles.
- **Normalization:** Since each triangle is counted three times (once per edge), total counts were divided by 3.
- **Per-Vertex Metrics:** Calculated triangles, degrees, and local clustering coefficients for each node.

Iterative Implementation in Spark

The computation used **PySpark DataFrame operations**:

- Joins were used to access adjacency lists for both vertices of each edge.
- Aggregations (`groupBy`, `sum`, `avg`) produced per-vertex and global statistics.

- The global clustering coefficient was computed as the ratio of closed triplets (triangles) to all possible triplets.

Performance and Discrepancies

- **Performance:** Joins on adjacency lists are shuffle-intensive; runtime grows with graph density. Works well for moderately large graphs but expensive for dense networks.
- **Accuracy:** Exact triangle counting ensures precise clustering metrics.
- **Discrepancies:**
  - High-degree nodes increase memory usage due to large neighbor sets.
  - Repeated joins and intersections may cause slight computational overhead.

Distance-Based Metrics

Algorithm Design Choices

Distance-based metrics describe how far apart nodes are in the graph, providing insights into connectivity and communication efficiency.

- **Graph Collection:** The undirected graph was collected to Python for in-memory traversal.
- **Adjacency Representation:** Constructed adjacency lists for efficient neighbor lookups.
- **Shortest Path Computation:** Used **Breadth-First Search (BFS)** from each node to compute shortest path lengths to all reachable vertices.
- **Metrics:**
  - **Diameter:** Maximum shortest-path distance across all node pairs.
  - **Effective Diameter:** 90th percentile of all pairwise distances, capturing typical reachability in the graph.

Iterative Implementation

Each vertex served as a BFS root, and its distance map was aggregated into a global list of shortest paths.
This all-pairs BFS method ensures accuracy for small to medium graphs but is computationally heavy for very large networks.

Performance and Discrepancies

- **Performance:** $O(V \times (V+E))$ complexity; feasible only for smaller graphs since all nodes perform BFS.
- **Accuracy:** Exact distances yield precise diameter estimates.
- **Discrepancies:**
  - Memory and runtime increase rapidly with graph size.
  - Collecting data from Spark to Python limits scalability.