Assignment -2- 2022594.

Part-I:- (A):

```
path = "/content/wiki-Vote.txt"

raw = spark.read.text(path)
lines = (raw
         .select(F.trim(F.col("value")).alias("line"))
         .filter((F.col("line") != "") & (~F.col("line").startswith("#"))))

parts = F.split(F.col("line"), r"\s+")
edges = (lines
         .select(parts.getItem(0).cast(T.LongType()).alias("src"),
                 parts.getItem(1).cast(T.LongType()).alias("dst"))
         .dropna()
         .dropDuplicates()
         .cache())

vertices = (edges
            .select(F.col("src").alias("id")).union(edges.select(F.col("dst").alias("id")))
            .distinct()
            .cache())

print("Edges:", edges.count())
print("Nodes:", vertices.count())
```

Ground Truth: 103689 : edges
7115 : Nodes

Computed values: Same as ground truth

- the wiki-vote file is read one line at a time... this dataset is a network repr. of voting interactions of wikipedia's admin election process... leading and trailing spaces are removed and col: value is renamed to line, empty lines are removed, and comment lines are removed.

modified format: "node1 node2"
"node3 node4"
:
"nodeX nodeY."

the lines are then split into node pairs. (0th index → src, 1st → dst) these are then converted to integer values, malformed rows are then removed, remove duplicate edges, and finally cache all values.

then construct a vertex list, and take unique src, dst ID's use .distinct to remove duplicate values, .cache to keep in memory.

finally use spark to print, no. of nodes and edges.

Part (B): we first configure spark for parallelism, partitions, and we also use adaptive execution for consistency... we then make undirected edges as WCC's are searched for in undirected graphs...

```
edges_sym = (
    edges
    .select("src","dst")
    .unionByName(edges.select(F.col("dst").alias("src"), F.col("src").alias("dst")))
    .distinct()
)

edges_ud = (
    edges_sym
    .where(F.col("src") != F.col("dst"))
    .withColumn("a", F.least("src","dst"))
    .withColumn("b", F.greatest("src","dst"))
    .select(F.col("a").alias("src"), F.col("b").alias("dst"))
    .distinct()
    .cache()
)
```

for ∀ src-dst ↴ add dst-src.

remove self loop.
↴ apply canonical ordering

we remove self loops (simple graph), canonical ordering is applied, finally we keep only single node pair as graph is undirected. (finally remove duplicates).

then labels are initialized, each vertex is given a unique ID to identify each component.

↓

Node could represent origin...

Set max_iterations to 20 or 10 or 5. make them, persistant, store in memory or disk...

```
labels = vertices.withColumn("label", F.col("id")).persist(StorageLevel.MEMORY_AND_DISK)
MAX_ITER = 20
```

In WCC goal is to group nodes together that can reach each other. we simply join vertices and give the smallest label ID...

↓

eventually all nodes converge.

```python
print("\n[Stage 1] Label Propagation (WCC)...\n")
for it in range(MAX_ITER):
    start = time.time()
    print(f"🔁 Iteration {it+1}/{MAX_ITER} ...")

    nbr_labels = (
        edges_bi
        .join(labels.withColumnRenamed("id","src_id"),
              F.col("src")==F.col("src_id"), "inner")
        .select(F.col("dst").alias("id"), F.col("label").alias("cand_label"))
        .persist(StorageLevel.MEMORY_AND_DISK)
    )

    new_labels = (
        labels.select("id","label")
              .unionByName(nbr_labels.withColumnRenamed("cand_label","label"))
              .groupBy("id").agg(F.min("label").alias("label"))
    )

    if (it + 1) % 5 == 0:
        new_labels = new_labels.checkpoint()
    else:
        new_labels = new_labels.persist(StorageLevel.MEMORY_AND_DISK)

    nbr_labels.unpersist(blocking=True)
    labels.unpersist(blocking=True)
    gc.collect()
    labels = new_labels

    print(f"    ↳ Iter {it+1} done in {time.time()-start:.2f}s")

print("\n✅ Label propagation finished.\n")
```

join src, dst to label(src)

select candidate label (join)

merge own lable and all candidate label

pick min label to represent...

checkpoint after 5 iterations. → clean

save lineage.

1→2, 2→3...

eg 2 to 1

1, 2, 3 → 1

sort by largest component...

After aggregating

Basically count the component sizes and sort...

finally we have two tables vertex, label_of_component, count number of edges in wcc. src, dst and

```
wcc = labels.cache()
wcc_sizes = wcc.groupBy("label").count().orderBy(F.desc("count")).cache()
largest_wcc_row   = wcc_sizes.first()
largest_wcc_id    = largest_wcc_row["label"]
largest_wcc_nodes = largest_wcc_row["count"]

print(f"✅ Largest WCC label = {largest_wcc_id}")
print(f"✅ Largest WCC nodes = {largest_wcc_nodes}")

w_ids = wcc.select("id","label")

edges_wcc_directed = (
    edges
    .join(F.broadcast(w_ids.withColumnRenamed("id","src_id").withColumnRenamed("label","lsrc")),
          edges.src == F.col("src_id"))
    .join(F.broadcast(w_ids.withColumnRenamed("id","dst_id").withColumnRenamed("label","ldst")),
          edges.dst == F.col("dst_id"))
    .where((F.col("lsrc") == largest_wcc_id) & (F.col("ldst") == largest_wcc_id))
    .select("src","dst")
    .cache()
)

largest_wcc_edges_directed = edges_wcc_directed.count()

print(f"✅ Largest WCC edges (DIRECTED) = {largest_wcc_edges_directed}  ← matches GT ≈ 103,663")

edges_wcc_undirected = (
    edges_ud
    .join(F.broadcast(w_ids.withColumnRenamed("id","src").withColumnRenamed("label","lsrc")), "src")
    .join(F.broadcast(w_ids.withColumnRenamed("id","dst").withColumnRenamed("label","ldst")), "dst")
    .where((F.col("lsrc")==largest_wcc_id) & (F.col("ldst")==largest_wcc_id))
    .select("src","dst").distinct()
)
```

↓ group in desc. order of component size

↑ rename to join src with dst, that belong to largest id.

↑ finally count number of edges in directed.

```
edges_wcc_undirected = (
    edges_ud
    .join(F.broadcast(w_ids.withColumnRenamed("id","src").withColumnRenamed("label","lsrc")), "src")
    .join(F.broadcast(w_ids.withColumnRenamed("id","dst").withColumnRenamed("label","ldst")), "dst")
    .where((F.col("lsrc")==largest_wcc_id) & (F.col("ldst")==largest_wcc_id))
    .select("src","dst").distinct()
)
print(f"ℹ️ Largest WCC edges (UNDIRECTED unique): {edges_wcc_undirected.count()}  [for sanity check]")

print("\nTop 10 WCCs by node count:")
wcc_sizes.show(10, truncate=False)


print("\n📊 Final Results — Weakly Connected Components")
print("-----------------------------------------------")
print(f"Total vertices: {vertices.count()}")
print(f"Total directed edges: {edges.count()}")
print(f"Largest WCC label: {largest_wcc_id}")
print(f"Largest WCC nodes: {largest_wcc_nodes}")
print(f"Largest WCC edges (DIRECTED): {largest_wcc_edges_directed}")
```

similarly for undirected

↳ print finally values.

Ground Truth: 7066 nodes
103,663 edges

Calculated values : same as Ground Truth.

Part C: Strongly Connected Components:

↓

nodes connected to every other and can reach each other.

first select directed edges in the graph ... with duplicates removed, create reverse edges...
Add node id's to each node pair, label to largest ie dst.

```
edges_dir = edges.select("src", "dst").distinct().cache()
edges_rev = edges_dir.select(F.col("dst").alias("src"), F.col("src").alias("dst")).cache()

vertices = (
    edges_dir.select(F.col("src").alias("id"))
    .unionByName(edges_dir.select(F.col("dst").alias("id")))
    .distinct()
    .cache()
)

print(f"✅ Graph loaded: {vertices.count()} vertices, {edges_dir.count()} directed edges.")


fwd_labels = vertices.withColumn("fwd_label", F.col("id")).persist(StorageLevel.MEMORY_AND_DISK)
MAX_ITER = 20
```

↓

Set forward propagation labels and store.

1 2 (2) ⇒
(2) 3 3 → label.

forward propagation loop → join source node with its table
create candidate label for each dst, combine node's own label to with candidates then aggregate.

```
print("\n[Stage 1] Forward reachability propagation...")
for it in range(MAX_ITER):
    start = time.time()
    print(f"  🔁 FWD Iteration {it+1}/{MAX_ITER} ...")

    new_fwd = (
        edges_dir
        .join(fwd_labels.withColumnRenamed("id", "src_id"),
              F.col("src") == F.col("src_id"), "inner")
        .select(F.col("dst").alias("id"), F.col("fwd_label").alias("cand_label"))
        .union(fwd_labels.select("id", "fwd_label").withColumnRenamed("fwd_label", "cand_label"))
        .groupBy("id").agg(F.min("cand_label").alias("fwd_label"))
    )

    if (it + 1) % 5 == 0:
        new_fwd = new_fwd.checkpoint()
    else:
        new_fwd = new_fwd.persist(StorageLevel.MEMORY_AND_DISK)

    fwd_labels.unpersist(blocking=True)
    gc.collect()
    fwd_labels = new_fwd
    print(f"     ↳ Done in {time.time()-start:.2f}s")

print("✅ Forward propagation complete.")
```

↓ join step by step.

→ set checkpoints.

Similar in backward direction. → compute smallest label...

finally combine fwd and bwd labels.

```
print("\n[Stage 3] Computing SCC intersection...")
scc_labels = (
    fwd_labels
    .join(bwd_labels, "id")
    .withColumn("label", F.concat_ws("_", F.col("fwd_label"), F.col("bwd_label")))
    .cache()
)

scc_sizes = (
    scc_labels.groupBy("label").count().orderBy(F.desc("count")).cache()
)

largest_scc_row = scc_sizes.first()
largest_scc_label = largest_scc_row["label"]
largest_scc_nodes = largest_scc_row["count"]

print(f"✅ Largest SCC label = {largest_scc_label}, nodes = {largest_scc_nodes}")

print("\n[Stage 4] Counting internal SCC edges...")

edges_with_labels = (
    edges_dir
    .join(scc_labels.withColumnRenamed("id", "src_id").withColumnRenamed("label", "src_label"),
          edges_dir.src == F.col("src_id"))
    .join(scc_labels.withColumnRenamed("id", "dst_id").withColumnRenamed("label", "dst_label"),
          edges_dir.dst == F.col("dst_id"))
    .select("src", "dst", "src_label", "dst_label")
    .cache()
)
```

↑ for each node. ↑ combine fwd and bwd label.

↑ find out sizes.

→ find largest scc label.

↳ count of nodes

→ find no. of directed edges in largest id.

↓

count edges.

and keep edges with same label.
for src and dst.

finally create a summary and print.

```
scc_edges = (
    edges_with_labels
    .filter(F.col("src_label") == F.col("dst_label"))
    .groupBy("src_label")
    .agg(F.count("*").alias("edges"))
    .cache()
)

scc_summary = (
    scc_sizes.join(scc_edges, scc_sizes.label == scc_edges.src_label, "left")
    .select(scc_sizes.label.alias("label"),
            scc_sizes["count"].alias("nodes"),
            F.coalesce("edges", F.lit(0)).alias("edges"))
    .orderBy(F.desc("nodes"))
    .cache()
)

largest_scc_edges = (
    scc_summary.filter(F.col("label") == largest_scc_label).select("edges").first()["edges"]
)

print(f"\n✅ Largest SCC → Nodes: {largest_scc_nodes}, Edges: {largest_scc_edges}")

print("\nTop 10 SCCs by node count:")
scc_summary.show(10, truncate=False)


print("\n📊 Final Results — Strongly Connected Components")
print("-------------------------------------------------")
print(f"Total vertices: {vertices.count()}")
print(f"Total directed edges: {edges_dir.count()}")
print(f"Largest SCC label: {largest_scc_label}")
print(f"Largest SCC nodes: {largest_scc_nodes}")
print(f"Largest SCC edges: {largest_scc_edges}")
```

Ground Truth : nodes : 1300
edges : 39,456

Calculated : same as GT.

Part D: firstly build a neighbor list.

create renamed version of undirected edges... and order as $u < v$ → Canonicalize.

join each node with both nodes neighbor list.

then find common neighbors between the two... take intersection.

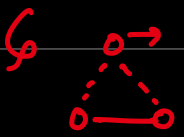
```
spark.conf.set("spark.sql.shuffle.partitions", "200")
spark.sparkContext.setCheckpointDir("/tmp/checkpoints")

nbrs = (
    edges_ud                                    # find neighbors
    .select(F.col("src").alias("id"), F.col("dst").alias("nbr"))
    .unionByName(edges_ud.select(F.col("dst").alias("id"), F.col("src").alias("nbr")))
    .groupBy("id")
    .agg(F.collect_set("nbr").alias("nbrs"))
    .persist(StorageLevel.MEMORY_AND_DISK)
)
print(f"Neighbor lists constructed for {nbrs.count()} nodes")

uv = edges_ud.select(F.col("src").alias("u"), F.col("dst").alias("v")).cache()

uv_nbrs = (                                     # join both neighbor lists for an edge.
    uv.join(
        nbrs.withColumnRenamed("id", "u_id").withColumnRenamed("nbrs", "u_nbrs"),
        uv.u == F.col("u_id"), "inner"
    )
    .join(
        nbrs.withColumnRenamed("id", "v_id").withColumnRenamed("nbrs", "v_nbrs"),
        uv.v == F.col("v_id"), "inner"
    )
    .select("u", "v", "u_nbrs", "v_nbrs")
    .persist(StorageLevel.MEMORY_AND_DISK)
)
print(f"Joined neighbor lists for {uv_nbrs.count()} edges")

@F.udf(T.ArrayType(T.LongType()))
def intersect_sorted(us, vs):
    if us is None or vs is None:                # → find intersection
        return []
    return sorted(list(set(us).intersection(vs)))
```

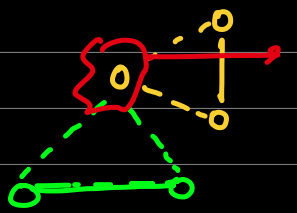


find common neighbors and explode to form triplets.

Condition — Canonicalized $u < v < w$ st no triangle counted thrice...

finally count total triangles.

count no. of triangles each node is in...



Count each nodes degree

Compute local clustering coeff. using.

$$C_i^0 = \frac{2T_i}{deg_i^0(deg_i^0 - 1)}$$

then find avg.

for closure ratio: count total triplets
↓
open triplets

∴ T → triangles
Σ → wedges → open triplets.
↓
∨ → wedge

finally display all values.

```
triangles_uv_w = (
    uv_nbrs
    .withColumn("w_all", intersect_sorted("u_nbrs", "v_nbrs"))
    .select("u", "v", F.explode("w_all").alias("w"))
    .where(F.col("u") < F.col("v"))
)
triangles = triangles_uv_w.where(F.col("v") < F.col("w")).distinct().cache()

total_triangles = triangles.count()
print(f"\n✅ Triangles (undirected simple): {total_triangles}")

tri_nodes = (
    triangles.select(F.col("u").alias("id"))
    .unionByName(triangles.select(F.col("v").alias("id")))
    .unionByName(triangles.select(F.col("w").alias("id")))
    .groupBy("id").agg(F.count("*").alias("t_i"))
    .persist(StorageLevel.MEMORY_AND_DISK)
)
print(f"Computed triangle participation for {tri_nodes.count()} nodes")

deg = (
    edges_ud.select(F.col("src").alias("id"))
    .unionByName(edges_ud.select(F.col("dst").alias("id")))
    .groupBy("id").agg(F.count("*").alias("deg"))
    .persist(StorageLevel.MEMORY_AND_DISK)
)
print(f"Computed degree for {deg.count()} nodes")
```

↑ intersection of neighbors
↓ form single row per neighbor
↓ unique Δ's.
↓ no. of Δs each node is in
↓ degree of each node

```
tc_deg = (
    vertices.select("id")
    .join(deg, "id", "left")
    .join(tri_nodes, "id", "left")
    .fillna({"deg": 0, "t_i": 0})
    .withColumn(
        "clust",
        F.when(F.col("deg") >= 2,
               (2.0 * F.col("t_i")) / (F.col("deg") * (F.col("deg") - 1)))
         .otherwise(F.lit(0.0))
    )
    .persist(StorageLevel.MEMORY_AND_DISK)
)

avg_clustering = tc_deg.agg(F.avg("clust")).first()[0]
print(f"\n✅ Average clustering coefficient: {avg_clustering:.6f}")

wedges = (
    tc_deg
    .withColumn("w", (F.col("deg") * (F.col("deg") - 1)) / 2.0)
    .agg(F.sum("w")).first()[0]
)
closure_ratio = (total_triangles / wedges) if wedges and wedges > 0 else None
transitivity = (3.0 * total_triangles / wedges) if wedges and wedges > 0 else None

print(f"✅ Closure ratio (T/τ): {closure_ratio}")
print(f"✅ Transitivity (3T/τ): {transitivity}")

print("\nSample local clustering coefficients:")
tc_deg.orderBy(F.desc("clust")).show(10, truncate=False)
```

⇒ Calculate local clustering
↑ Avg. clustering
↓ count triplets (open)
↓ find closure ratio.

Part E: Diameter and 90% diameter

take all vertices belonging to largest wcc.

↓

only dist in main component.

Keep edges with both endpoints in largest component.

Build adjacency lists, node, neighbors pair similar to before.

gather all nodes in memory for BFS as a set.

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", "-1")  # avoid surprise broadcasts
spark.conf.set("spark.sql.shuffle.partitions", "96")
spark.sparkContext.setCheckpointDir("/tmp/checkpoints")
sc = spark.sparkContext

lcc_nodes = (                                   ↑ find all nodes in largest component
    wcc.filter(F.col("label") == largest_wcc_id)
       .select(F.col("id").alias("id"))
       .cache()
)

E_lcc_df = (                                    ↑ Keep only edges completely in LCC.
    edges_bi
    .join(lcc_nodes.withColumnRenamed("id","src_id"), F.col("src")==F.col("src_id"))
    .join(lcc_nodes.withColumnRenamed("id","dst_id"), F.col("dst")==F.col("dst_id"))
    .select("src","dst")
    .distinct()
    .cache()
)

nodes_cnt = lcc_nodes.count()                   → count nodes and edges in LCC
edges_cnt = E_lcc_df.count()                    →
print(f"LCC edges: {edges_cnt}, nodes: {nodes_cnt}")

adj = (
    E_lcc_df
    .rdd                                        → create adj. list.
    .map(lambda r: (int(r['src']), int(r['dst'])))
    .groupByKey()                               # (src, <iterable dst>)
    .mapValues(lambda it: list(set(it)))        # dedup neighbors
    .persist(StorageLevel.MEMORY_AND_DISK)
)
adj.count()  # materialize

all_nodes = set([int(r.id) for r in lcc_nodes.select("id").collect()])
```

↳ collect all nodes for BFS.

Perform BFS.

calculate distances from one node to other and store.

result is a dictionary with src and dst distance

eg for node 5 ⇒ {1: 153,
2: 253...}

for each of random nodes run BFS.
find max diameter and
and no. of node pairs / distance.

eg: 5000 nodes at 1 dist.. ~

compute cdf and 90th percentile distance. (diameter).

final Comparisions of GT vs Computed values:

Summary —

Metric Comparison Summary:

Metric	GroundTruth	Computed	AbsDiff	RelDiff_%
Nodes	7115.0000	7115.0000	0.0000	0.0000
Edges	103689.0000	103689.0000	0.0000	0.0000
Largest WCC (nodes)	7066.0000	7066.0000	0.0000	0.0000
Largest WCC (edges)	103663.0000	103663.0000	0.0000	0.0000
Largest SCC (nodes)	1300.0000	1300.0000	0.0000	0.0000
Largest SCC (edges)	39456.0000	39456.0000	0.0000	0.0000
Avg. clustering coefficient	0.1409	0.1409	-0.0000	-0.0015
Number of triangles	608389.0000	608389.0000	0.0000	0.0000
Fraction of closed triangles	0.0456	0.0418	-0.0038	-8.3559
Diameter	7.0000	6.0000	-1.0000	-14.2857
Effective diameter (90-percentile)	3.8000	3.7382	-0.0618	-1.6275

Comparitive graph —