Advanced Human-Machine Interaction
Interaction Data Analysis
TD05: Social Network Analysis

INSA Rouen Normandie - Normandie University

Objectives. Formulate a graph analysis problem, distinguish node roles from communities, run and interpret Louvain, and implement its aggregation phase with consistent weights.

Material. `student.py`, `helpers.py`, `requirements.txt`. Python 3.12 or newer (excluding 3.14.1). Install dependencies before class:

```
python -m pip install -r requirements.txt
python student.py
```

The graph is bundled with NetworkX; no dataset download or account is needed. The helpers provide visualisation and label-independent partition comparison. Complete the TODOs in the starter progressively.

1. From social relationships to a graph

We use Zachary's karate club network: 34 members and their observed social ties. This historical network describes offline relationships, not platform logs. The same analysis pipeline can be applied to mediated interactions, provided the graph construction is justified.

1. Inspect `G.number_of_nodes()`, `G.number_of_edges()`, `G.is_directed()`, and `nx.is_connected(G)`. Inspect one node and one edge with `G.nodes[0]` and `G[0][1]`.
2. State what a node and an edge represent. Edge weight records the number of contexts in which two members interacted. What information is lost when we ignore weights? What temporal information is absent?
3. For a network built from online replies, propose a node definition, an edge direction, a weight and an observation window. Would the undirected model used here retain all relevant information?

The node attribute `club` records membership after the eventual split (`Mr. Hi` or `Officer`). Use it only for interpretation, never as an input to community detection.

2. Identify structurally different users

For the core exercises, treat the graph as **unweighted**. A large interaction count is not a large distance.

1. Compute the following scores:
   `nx.degree_centrality(G)`,
   `nx.closeness_centrality(G)`,
   `nx.betweenness_centrality(G, weight=None)`.
2. For each measure, report the three highest-scoring nodes and their scores. Use ascending node ID to break ties.
3. Explain one difference between these rankings. Which measure describes many direct contacts? Short paths to other users? A position on shortest paths connecting others?
4. Suggest one hub, one potential broker and one peripheral member, citing structural evidence. Are these exclusive categories? Can these measures prove influence or actual information flow?

If ahead: compare with `nx.eigenvector_centrality(G, weight=None)`.

3. Detect and interpret communities

```
P = nx.community.louvain_communities(G, weight=None, seed=0)
Q = nx.community.modularity(G, P, weight=None)
plot_partition(G, P, 'louvain.png')
```

1. Report the communities, their sizes and modularity. What is the algorithm trying to optimise? Does it guarantee the global optimum?
2. Compare with the observed split: build a small table with one row per detected community and two columns counting its `Mr. Hi` and `Officer` members. Does the partition coincide with the split? Explain why it need not contain exactly two communities.
3. Give one limitation of interpreting a structural community as a social group.

If ahead: repeat with `weight='weight'` in *both* calls. Interpret changes as a change of graph model; modularity values from different weight models are not a direct accuracy comparison.

4. Implement the aggregation phase

Implement `aggregate_graph(G, partition, weight=None)`. You are **not** asked to implement the local-move phase or the complete Louvain algorithm.

Input: an undirected graph and a list of nonempty sets partitioning its nodes.
Output: an undirected graph H, with community IDs $0, 1, \ldots$ as nodes, and an edge attribute `weight` storing the sum of original edge weights. Internal edges become self-loops.

1. Create a dictionary mapping each original node to its community ID; add all supernodes, including isolated ones.
2. Iterate over `G.edges(data=True)`. NetworkX returns each undirected edge once. Read weight 1 when `weight=None`, otherwise the selected attribute (default 1).
3. Add this weight to the edge between the corresponding supernodes; the endpoints may be identical. Create the edge if it does not exist yet.

First test. `toy_graph()` contains triangles $\{1, 2, 3\}$ and $\{4, 5, 6\}$, joined by edge $(3, 4)$. All seven edges have unit weight. Aggregate using these two sets. Predict the two loop weights, the connecting edge weight and the total edge weight before running your code.

Then apply to karate. Aggregate the partition from Part 3, keeping `weight=None` for the original graph. Check:

$$m(G) = m(H), \qquad s_H(C) = \sum_{u \in C} s_G(u).$$

Use `G.size(weight=None)` and `H.size(weight='weight')`; for strengths use `G.degree(u, weight=None)` and `H.degree(c, weight='weight')`. Compare the original modularity with that of the **singleton partition of** H using `weight='weight'` for H. They should agree up to rounding (e.g. 10^{-10}). This checks aggregation, not another optimisation step.

Counting rule. A loop of weight w contributes w to total edge weight and $2w$ to node strength. Do not double its stored weight.

5. Is the result stable?

Run unweighted Louvain for seeds 0 through 9, keeping resolution at its default value 1. Report seed, number of communities and modularity. Compare partitions with `canonical_partition(P)`, which ignores community numbering. Are the partitions identical? Do similar modularity scores imply identical memberships? Report your observation even if all runs happen to agree.

Additionnal work

1. Finish the core tasks and prepare a concise report (about 2 pages, excluding code): centrality table and interpretation, community comparison, aggregation checks, and stability table.
2. Generate 20 partitions with seeds 0 through 19 at fixed resolution 1 and fixed `weight=None`. Implement `consensus_matrix(partitions, nodes)`:

$$M_{ij} = \frac{1}{R} \sum_{r=1}^{R} \mathbf{1}\{i, j \text{ belong to the same community in run } r\}.$$

   Use `nodes=list(G)` to define row and column order. Each run contributes 1 for a pair in the same community, otherwise 0. Divide by the number of runs. Plot with `plot_consensus(M, nodes, 'consensus.png')`.
3. Check symmetry, diagonal entries equal to 1, and entries in $[0, 1]$. Identify one consistently grouped pair and, if present, a pair with intermediate frequency. Interpret $M_{ij} = 0$, 0.5, and 1. Does this matrix itself define a final partition?
4. *Optional extension:* separately compare resolutions $0.5, 1, 2$, with seed 0 fixed. Pass the same resolution to the modularity calculation. Distinguish algorithmic variability (seed) from a change in the objective (resolution). Do not select a resolution merely by comparing these different objective values.

Submit: your completed Python file, a short report, and the main plots. State Python/NetworkX versions, seeds, weight convention and resolution. The reference correction will be distributed at the next practical session.

Annex: conventions and useful API calls

For a weighted undirected graph and resolution 1:

$$Q = \sum_{C} \left[\frac{w_C}{m} - \left(\frac{s_C}{2m} \right)^2 \right].$$

Here m is total edge weight counted once, w_C is internal edge weight counted once, and s_C is the sum of strengths of nodes in C. A self-loop contributes twice to strength. After aggregation, use the stored edge weights, even when the original graph was treated as unweighted.

```
# P is a list of sets; H is the aggregated graph.
print([len(c) for c in P])
print(list(H.edges(data=True)))
singletons = [{v} for v in H]
Q_H = nx.community.modularity(H, singletons, weight='weight')
# Provided ranking pattern (scores is a node -> score dictionary):
top3 = sorted(scores.items(), key=lambda p: (-p[1], p[0]))[:3]
# Label-independent comparison:
same = canonical_partition(P1) == canonical_partition(P2)
```

References. W. W. Zachary (1977), *An information flow model for conflict and fission in small groups.* V. D. Blondel et al. (2008), *Fast unfolding of communities in large networks*, doi:10.1088/1742-5468/2008/10/P10008. NetworkX documentation: https://networkx.org/documentation/stable/.