Twitter Community Finding

Jevon, Zahara, Marcos, and Dominik

Introduction

Twitter is a massive network where information spreads, ideas gain traction, and influence shapes behavior. Community detection is a fundamental tool for understanding these complex networks.

Identifying communities helps us make sense of large systems by revealing their hidden structure. For social networks, community detection can uncover groups of users who interact frequently, share interests, or influence one another. This has powerful uses: analyzing information flow and influence, detecting echo chambers, applying recommendation systems, and even spotting coordinated attacks/behavior.

Table of Contents

Leiden Algorithm: What is it?

Leiden is a **community detection** algorithm.

Goal: identify groups or communities of nodes that are more densely connected to each other than the rest of the network.

The algorithm **maximizes the quality function** or modularity in this case, effectively ensuring that the communities are well-connected and meaningful.

Quality Metric

Assesses the strength of division of a network into modules

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - \gamma \frac{k_i k_j}{2m}) \delta(c_i, c_j)$$

Q: Modularity score of the partitioning of the graph

m: number of edges of the graph or the total weight of all edges in the graph

Aij: adjacency score between node i and j. For an unweighted graph, 1 if there is an edge between i and j, 0 otherwise. For a weighted graph, the weight of the edge between i and j.

γ: resolution parameter (= 1 by default); controls size of the communities

ki, kj: degree of nodes i and j. For an unweighted graph, ki is the number of edges with node i; for a weighted it is the sum of the edge weights with node i.

δ(ci, cj): binary score (= 1 if nodes i and j are in the same community, = 0 otherwise)

Summary

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - \gamma \frac{k_i k_j}{2m}) \delta(c_i, c_j)$$

-Measures the difference between actual edge weight and expected edge weight in a random graph with the same node degrees.

-Done only for node pairs in the same community.

-Checks if more connections happen within communities than expected by chance.

-Score ranges from -0.5 to 1.

Three Phases:

Local Movement

- All nodes assigned to singleton communities
- Nodes are moved between neighbor communities to improve quality of partition
- **Goal:** improve modularity

Refinement

- Nodes are moved within communities to ensure maximum connectivity within modules
- Some communities will split into smaller, more well-connected communities when refined

Aggregation

- Communities from refinement are condensed into super-nodes
- The process repeats on the simplified network structure
- Continues until the quality function is optimized.

How it works

Local Movement

{A, B, C, D, E, F, G}

{A,B} → {A, B, C}

{D, E, F, G}

Refinement

Let’s say G is not strongly connected.

New Partition:
{A, B, C} {D, E, F} {G}

Aggregation

Node 1 ={A, B, C}

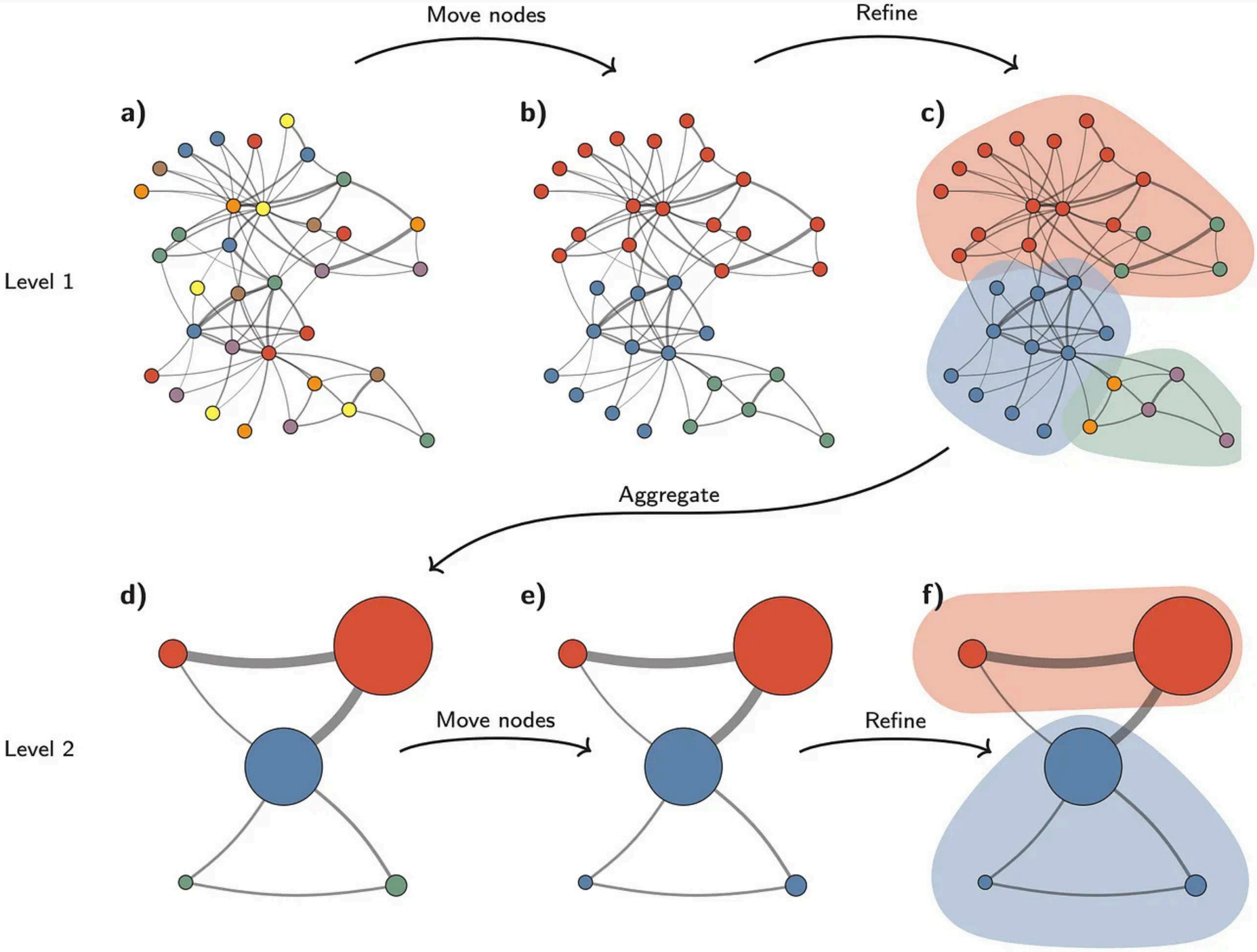
Node 2 = {D, E, F}

Node 3 = {G}

Weak edge between Nodes 2 and 3

In the Twitter network, users who are friends are grouped in the first phase. Next, we check if these groups are tightly connected. Lastly, we compress the network and repeat. Leads to high-quality communities based on users that form close social circles.

Visual


Move nodes
Refine
a)
b)
c)
Level 1
Aggregate
d)
e)
f)
Level 2
Move nodes
Refine

Our Implementation vs. IGraph Leiden Clustering

Time and Space complexity:

- Time: O(L|E|), where L is the total number of iterations performed
- Space: O(|V| + |E|)

Resolution Parameter:

- IGraph and other implementations use an optimiser function and try out different resolution parameters to select the best option for preserving the community structure/granularity of the graph and increasing modularity.
- To achieve a similar level of optimization, our program runs a loop to check how many iterations of the algorithm provide the highest modularity and therefore optimal community structure.

Notes:

- More iterations don’t equal Improved Modularity
- Information is “lost” with the compression of the graph
- Our implementation is not highly optimized and runs significantly slower especially for larger inputs :(

Infomap Algorithm – A Different Approach

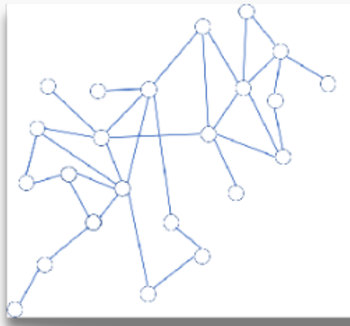
- Take our twitter graph, where nodes are users and directed edges are who users follow.
- We ask, ‘what are the communities on Twitter?’ But this is an **abstract question.**
- So, Infomap **reduces community detection** into a **data compression problem**
- This lets us define communities **objectively** - but how?

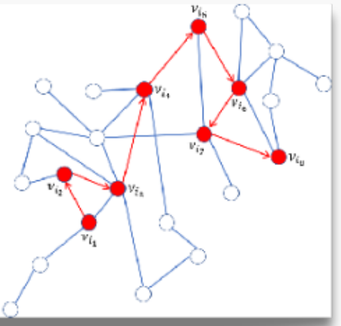
How it works

- **Place a random walker on our graph (Obama)**
- **The walker is more likely to stick in dense interconnected areas** (Where users follow each other)
  - Can think about this with common probability; A node with 8 internal edges and 2 exit edges has 80% chance of staying in the dense area.
- **Infomap observes the walker - which areas are the walker ‘sticking’ in? These are our communities**
  - But how long do we observe the walker? Do we have to visit every single node?

Random Walk

Community





How it doesn’t work

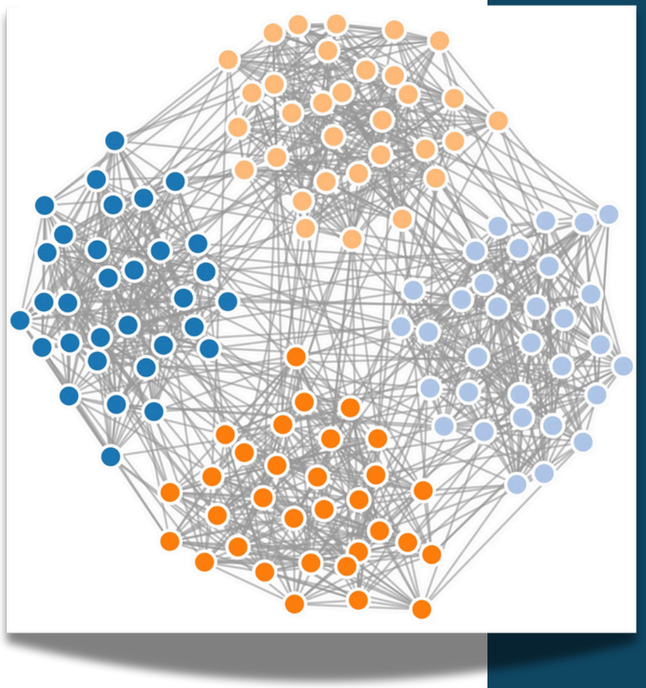
- Infomap doesn’t actually walk through the graph step-by-step, would be way too expensive
- Instead, it uses **probability models** to calculate the random walker:
  - Probability of **being on each node** in the long run (stationary distribution)
  - Probability of **staying** within a community
  - Probability of entering a new community (**exiting**)
- Then, it uses this **flow** to design a coding scheme: shorter codes for common transitions, longer codes for rare jumps
- The goal: **minimize** the average description length of the walker’s path (in bits)

Compression: The Objective Truth

- Imagine we want to **describe the path of the walker** as it moves across our graph.
- If we describe each node visit individually, it takes a **lot of information**.
    - **ex: obama -> biden -> kamala -> michelle -> obama**
- But since we know the walker sticks in certain communities for a while, we can compress better:
- First, say which community you're in. Then, use shorter codes to describe movements within that community
    - **ex: Community ‘Blue’: 0 -> 1 -> 2 -> 3 -> 0**
- When moving to a new community, we switch the codebook and can start mapping again.

Better Compression = Better Communities

- If the random walker can **stay in a community** without jumping out too often, it **takes less information** to describe their path!
- Infomap tries different ways of grouping nodes into communities.
- For each grouping, it asks, **How efficiently could we describe the walk?**
- This is measured using the Map Equation: giving us the total codelength in bits.
- It iterates to find and return the grouping that gives the shortest codelength
- **Essentially: The grouping with the shortest codelength is best compressed, and therefore, the best community structure**



How to Compress? The Map Equation

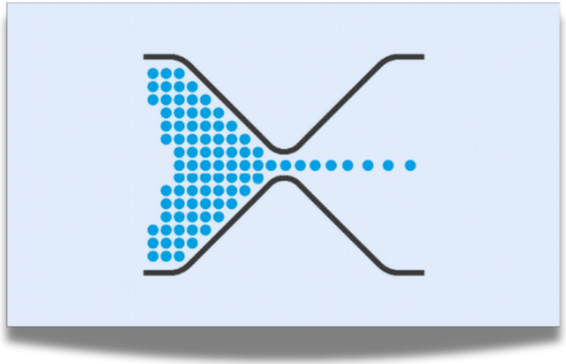
- Infomap aims to minimize L, the total codelength, by balancing:
  - How often the walker leaves communities q↷ (total_exit)
    - less jumps = less bits
  - How uncertain (entropy) inter-community jumps are (H_Q)
    - meaningful communities have predictable exits
  - The cost of describing movement inside communities (H_P)
    - communities should be strongly connected (i.e not jumping around too far)
- **Smaller/Shorter L = better compression = better communities**

$$L(M) = q_{\curvearrowright} H(\mathcal{Q}) + \sum_{i=1}^{m} p^i_{\circlearrowright} H(\mathcal{P}^i)$$

```
L ≈ total_exit * H_Q + H_P
```

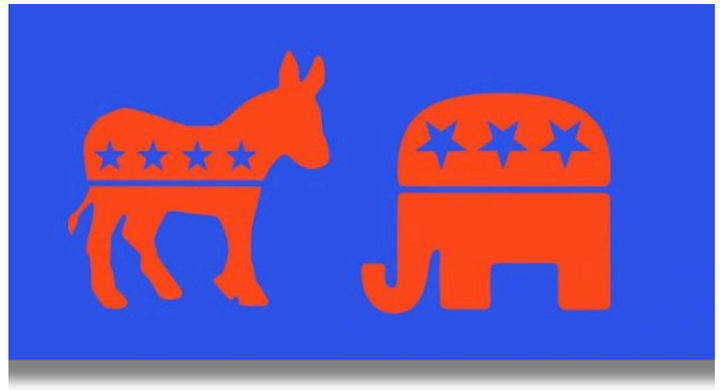
Our Implementation – Caveats

- Infomap is super optimized, uses heuristics, and does not have a strict worst case time complexity
  - Runs close to O(nlogn) where n = nodes.
- Runs entire 80,000 node, 1.2million edge graph in 30 seconds
- **ScratchInfomap is much slower** - can't finish full graph in under 24 hours
  - Loop over every node n for **O(n)** time
  - For every node, look at neighbors, try moves, and recalculate map equation - roughly **O(n)**
  - Do this for i iterations, can approximate as **O(i * $n^{2)}$ time**
- **Limitation:** Can only run on small subgraphs



Experiments

- Combined subgraphs centered around left-leaning and right-leaning news accounts
  - **CNN, The New Yorker, Fox News**

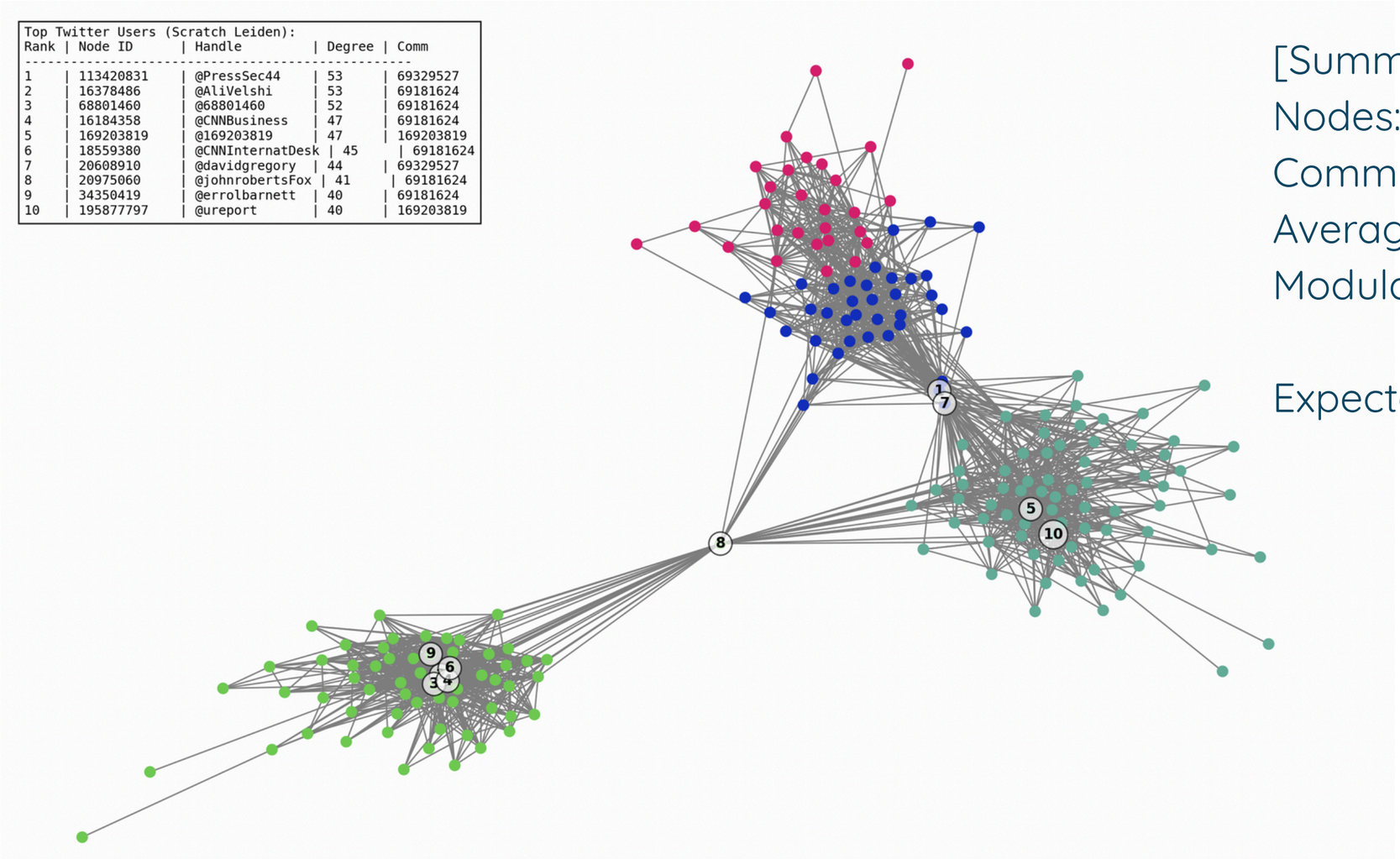


- **Goal 1:** Compare partioning between Leiden and Infomap communities
  - Are we getting **meaningful communities?**
- **Goal 2:** See the extent of which democrats and conservatives interact with each other
  - Do we detect **echo chambers?**

Our Implementation – ScratchLeiden

Top Twitter Users (Scratch Leiden):

Rank	Node ID	Handle	Degree	Comm
1	113420831	@PressSec44	53	69329527
2	16378486	@AliVelshi	53	69181624
3	68801460	@68801460	52	69181624
4	16184358	@CNNBusiness	47	69181624
5	169203819	@169203819	47	169203819
6	18559380	@CNNInternatDesk	45	69181624
7	20608910	@davidgregory	44	69329527
8	20975060	@johnrobertsFox	41	69181624
9	34350419	@errolbarnett	40	69181624
10	195877797	@ureport	40	169203819



[Summary]

Nodes: 200, Edges: 1669

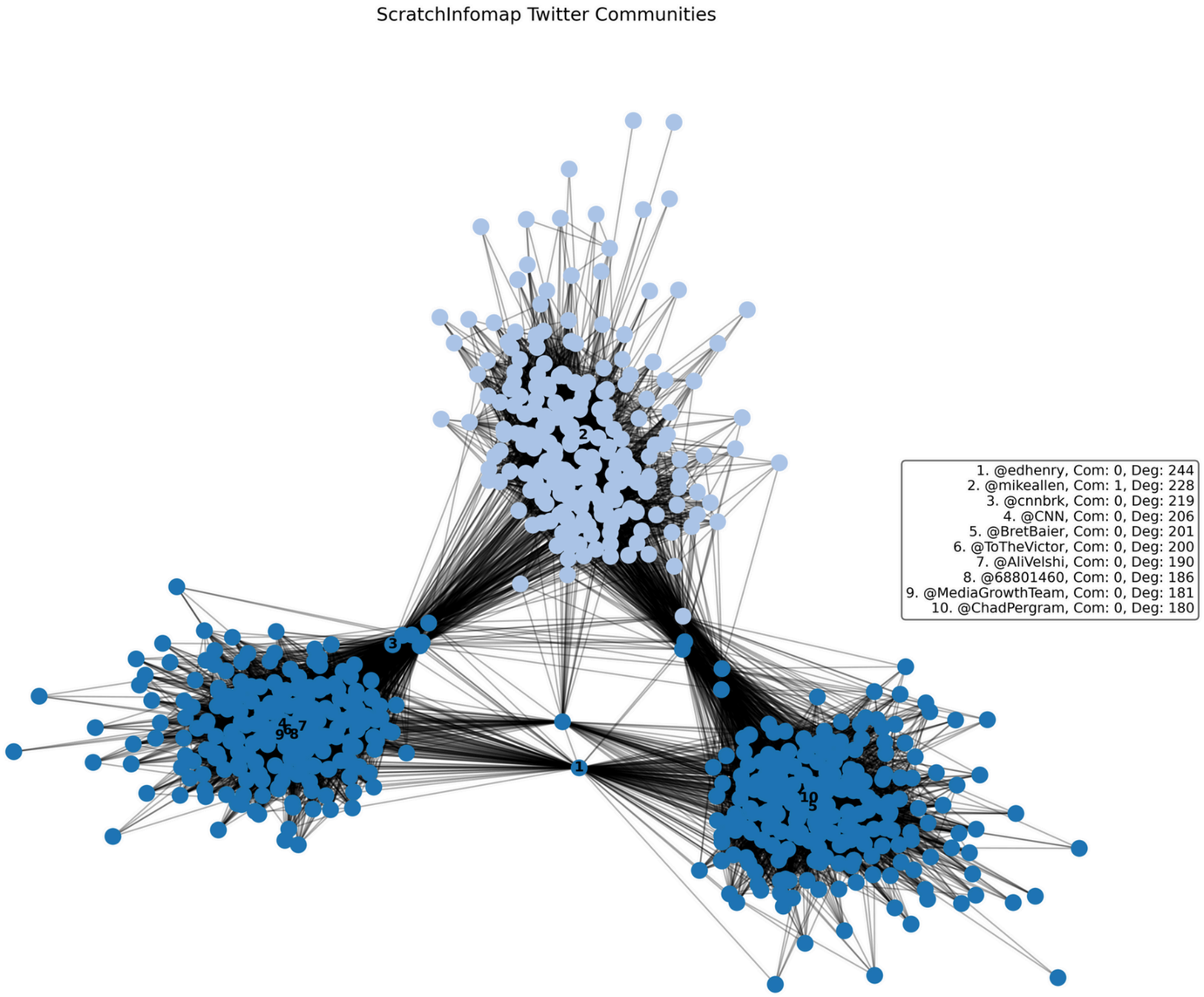
Communities found: 4

Average community size: 344.00

Modularity: 0.1657

Expected: 3 distinct communities

Our Implementation – ScratchInfoMap



[Summary]
Nodes: 688, Edges: 19397
Communities found: 2
Average community size: 344.00
Modularity: 0.3626

Expected: 3 distinct communities

Official Infomap finds 5 communities with average size 136,

Observation: Suggests Infomap's random walks followed high-flow connections between CNN and Fox - meaning many of these users probably weren't in strict echo chambers

Our Implementation – ScratchInfomap

- Trimming nodes to 200 max nodes allows ScratchInfomap to find the communities we expect to see
  - Shows that our implementation **struggles with higher complexity** graphs

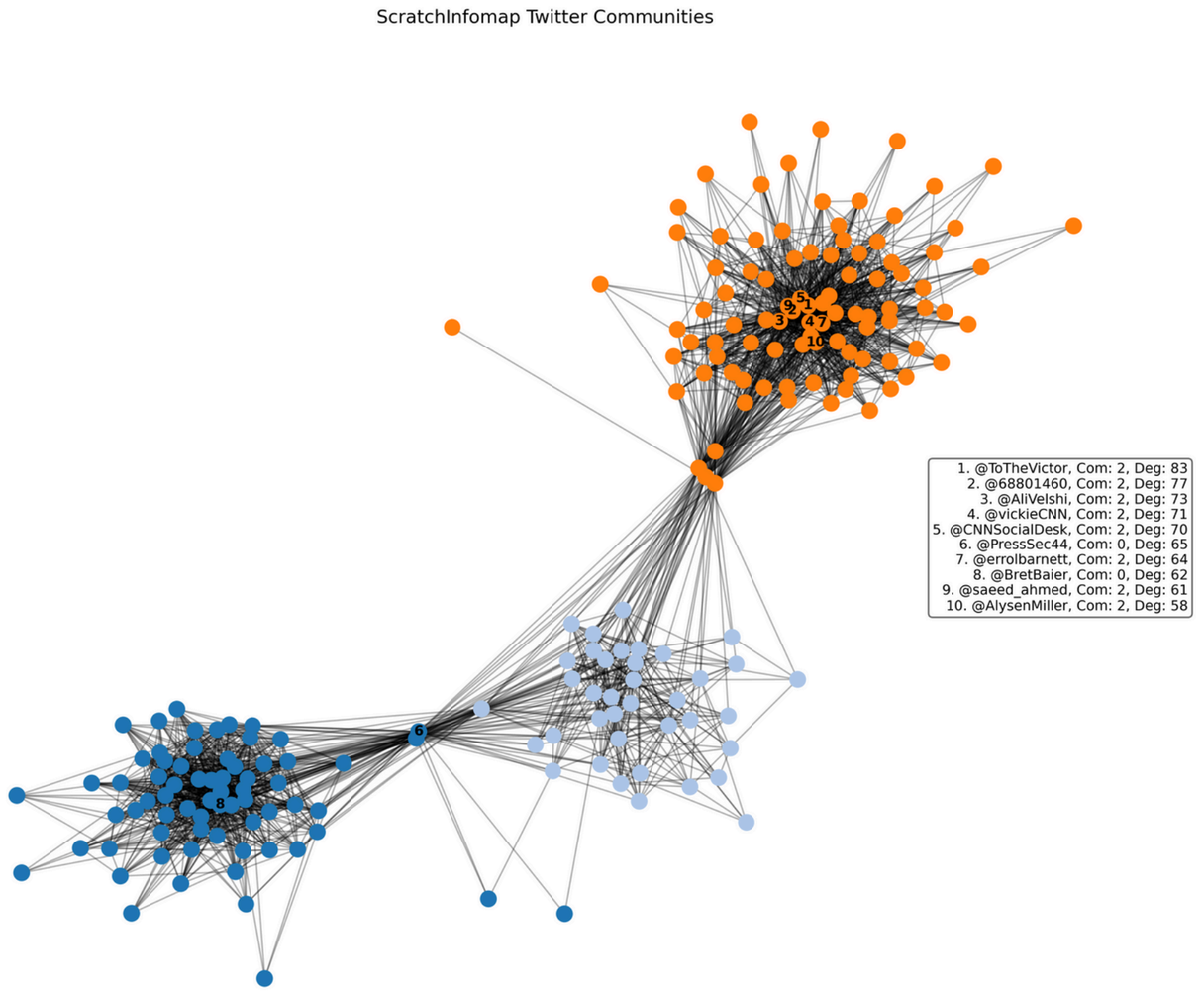
[Summary]

Nodes: 200, Edges: 2431

Communities found: 3

Average community size: 66.67

Modularity: 0.5482



Conclusion

- **Infomap** uses flow compression with the goal of minimizing the map equation.
- **Leiden** uses modularity optimization
- Both are extremely fast and good at what they do
- Would most likely use Leiden for full G
    - Infomap in theory would work better with weighted edges i.e graphs with interaction data like reposts and comments

If we had more time...

- Try to run ScratchInfomap on full G (would probably take weeks lol)
- Improve ScratchInfomap map equation - more random starts, caching, recursion
- Look for more users that bridge communities
- Optimize ScratchLeiden to have faster runtime, include more randomization, and improve overall performance.

Sources

Database:

https://snap.stanford.edu/data/

Sources:

https://medium.com/@swapnil.agashe456/leiden-clustering-for-community-detection-a-step-by-step-guide-with-python-implementation-c883933a1430https:/northernprotector.medium.com/modularity-score-6019955f0580
https:/www.nature.com/articles/s41598-019-41695-z#:~:text=The%20Leiden%20algorithm%20consists%20of,partition%20for%20the%20aggregate%20network.
https://dl.acm.org/doi/fullHtml/10.1145/3673038.3673146
https://en.wikipedia.org/wiki/Leiden_algorithm
https://www.pnas.org/doi/10.1073/pnas.0601602103
https://www.mapequation.org/assets/publications/mapequationtutorial.pdf
https://mapequation.github.io/infomap/index.html
https://www.statworx.com/en/content-hub/blog/community-detection-with-louvain-and-infomap

Thank you