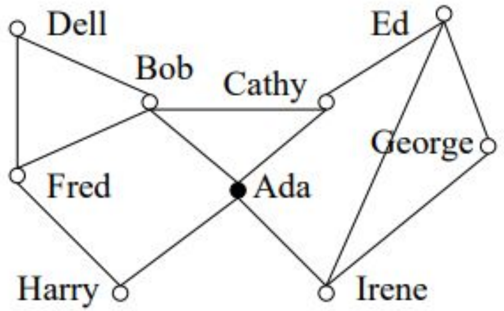
Preserving Privacy in Social Networks Against Neighborhood Attacks

Federico Conti
Nicolas Falcone

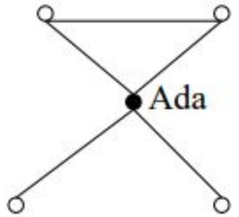
Neighborhood Attack

- Removing node and edge labels does not protect privacy.
- Having information about neighbors of a target victim and the relationship among the neighbors, it is possible to re-identify the target victim in an anonymized network.
- Using neighborhood attack, it is possible to analyze the connectivity of the target node and its relative position in the network.

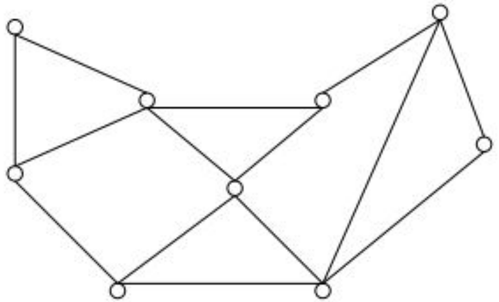


(a) the social network

Privacy can be provided by using the k – anonymity mode



(c) the 1−neighborhood graph of Ada



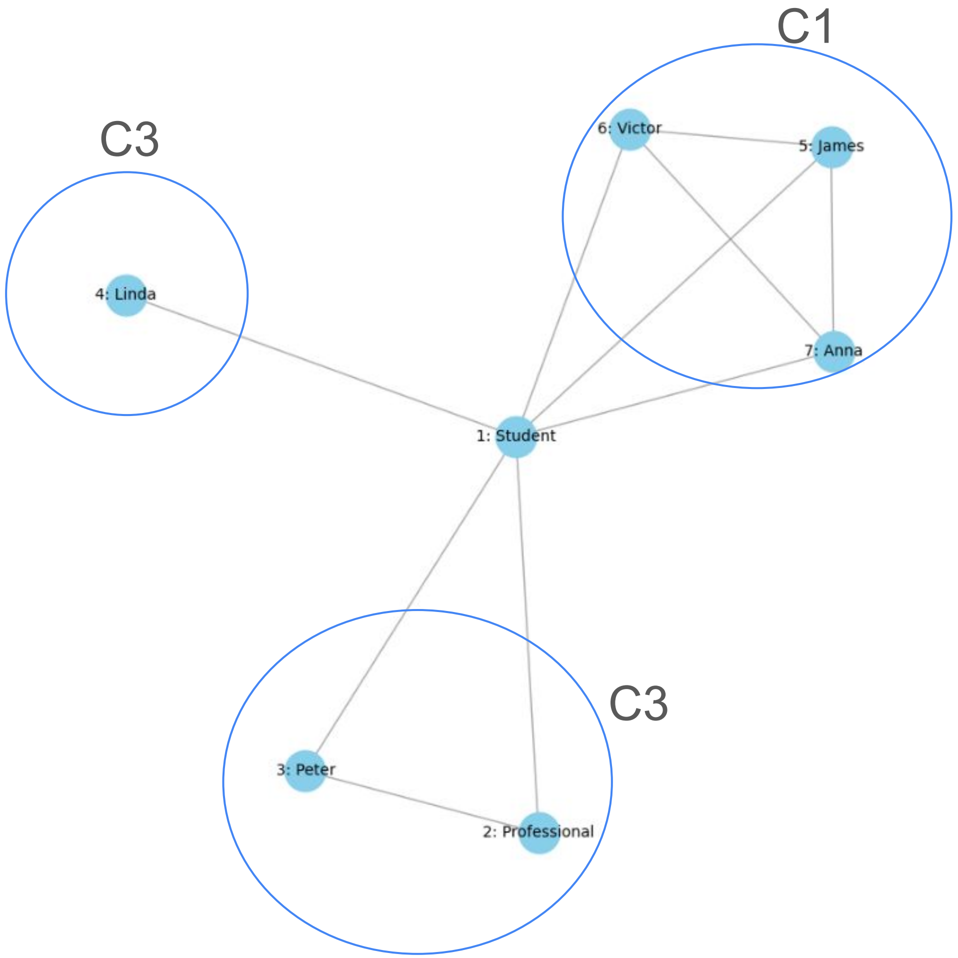
(d) privacy−preserved anonymous network

Neighborhoods extraction

For each vertex *u* in the social network *G*, extract its neighborhood, denoted by *NeighborG(u)*, which is the induced subgraph of *u*'s neighbors.

Workflow:

- **Neighborhood Component Decomposition:** Divide each neighborhood *NeighborG(u)* into its maximal connected subgraphs, called neighborhood components. (*C1, C2, and C3.*)
- **DFS Code Generation:** For each neighborhood component *C*, generate its best Depth-First Search (DFS) code to solve the uniqueness problem.
- **Neighborhood Component Code Construction:** Combine the minimum DFS codes of all components in a neighborhood to create the neighborhood component code (NCC) for the vertex *u*.



DFS representation

Goal: find a best representation to ensure matching during a component anonymization.

For each neighborhood component *C*, generate its Depth-First Search (DFS) code as follows:

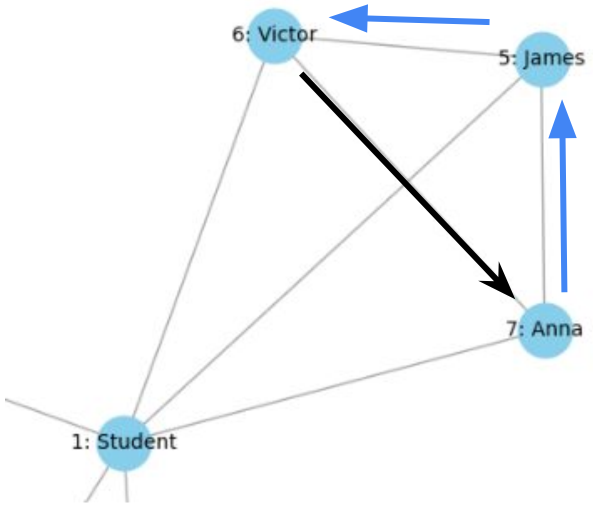
- Choose a starting vertex and perform a depth-first traversal of the component *C*.
- Represent each edge as (*vi, vj, L(vi), L(vj)*), where *vi* and *vj* are the vertices of the edge, and *L(vi)* and *L(vj)* are their corresponding labels.
  List all edges in the order ≺, which prioritizes **forward edges** (edges in the DFS tree) over **backward edges**, and within each category, sorts based on vertex indices.

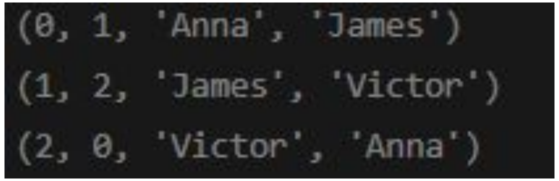
$F: \{(v_0, v_1), (v_1, v_2)\}$

$\le \quad \le$

$B: \{(v_2, v_0)\}$

$R: \{(v_0, v_1), (v_1, v_2), (v_2, v_0)\}$





```
(0, 1, 'Anna', 'James')
(1, 2, 'James', 'Victor')
(2, 0, 'Victor', 'Anna')
```

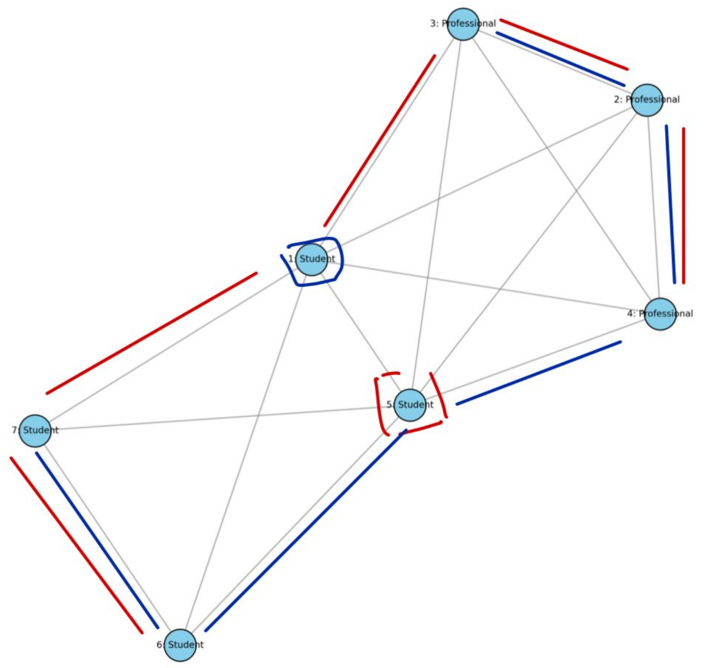
Neighborhoods Anonymization

K=2 for G=(1,5)

The overall goal is to make any individual vertex indistinguishable from at least *k*-1 others in the anonymized network, thereby limiting the confidence of an adversary performing a neighborhood attack.

Workflow:

Anonymization Cost: A cost function is defined to measure the information loss incurred during anonymization. This cost considers label generalization (using a normalized certainty penalty) and edge additions.
Greedy Anonymization: The algorithm processes vertices in descending order of neighborhood size, aiming to minimize information loss for high-degree vertices.
Vertex Grouping: For each "seed" vertex, *k*-1 other vertices with the most similar neighborhoods (lowest anonymization cost) are identified and grouped together.
Neighborhood Isomorphism: The neighborhoods within each group are made isomorphic.

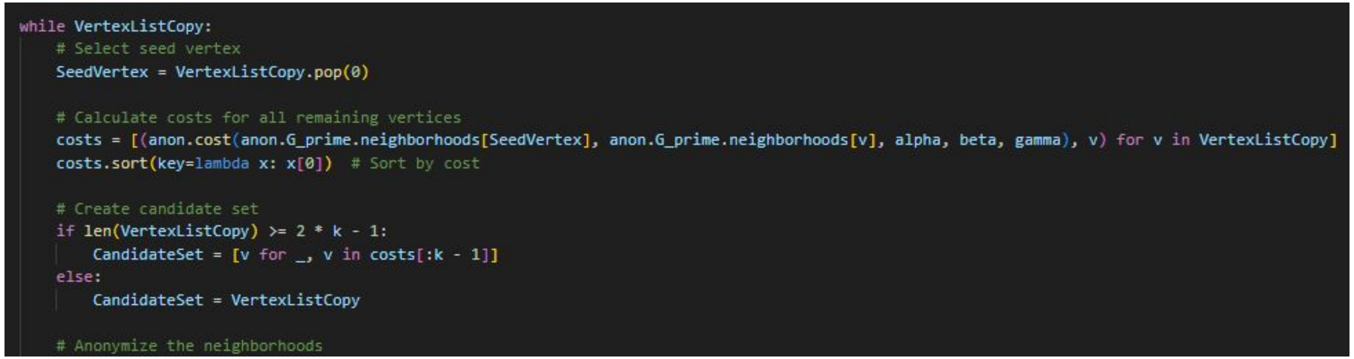


Anonymization Cost

A cost function measures the similarity between the neighborhoods of two nodes in a social network. The smaller the cost function, the more similar the two neighborhoods are.

The cost function is calculated using the following parameters:

- **α**: This weight is applied to the normalized certainty penalty (NCP), which quantifies the information loss resulting from generalizing the labels of vertices.
- **β**: This weight is associated with the information loss due to adding edges to the social network.
- **γ**: This weight is assigned to the number of vertices linked to the anonymized neighborhoods to achieve k-anonymity



```
while VertexListCopy:
    # Select seed vertex
    SeedVertex = VertexListCopy.pop(0)

    # Calculate costs for all remaining vertices
    costs = [(anon.cost(anon.G_prime.neighborhoods[SeedVertex], anon.G_prime.neighborhoods[v], alpha, beta, gamma), v) for v in VertexListCopy]
    costs.sort(key=lambda x: x[0])  # Sort by cost

    # Create candidate set
    if len(VertexListCopy) >= 2 * k - 1:
        CandidateSet = [v for _, v in costs[:k - 1]]
    else:
        CandidateSet = VertexListCopy

    # Anonymize the neighborhoods
```

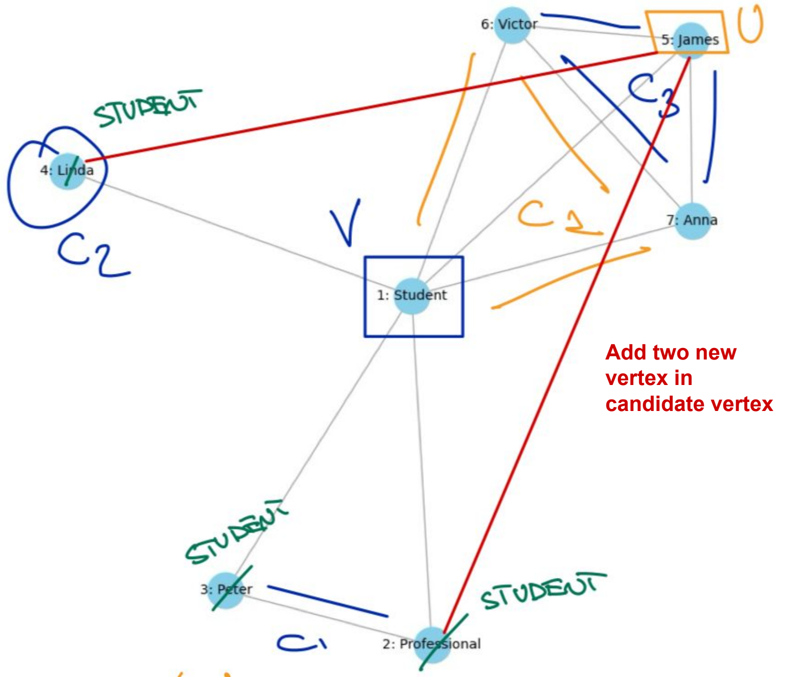
Neighborhoods Isomorphism

#comp_v = 2 > #comp_u = 0

Two components perfectly match each other if they have the same minimum DFS code

Steps

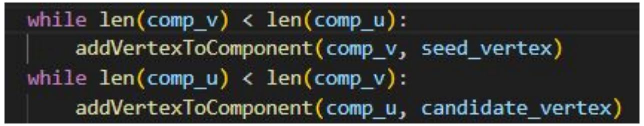
- **Add vertex to component**: Ensure both components have the same number of vertices by repeatedly adding vertices to the smaller component.
- **Label Generalization:** For any pair of matched vertices that have different labels, apply a label generalization technique to assign a common generalized label to both nodes.
- **Edge Alignment**:Compare the two adjacency matrices (A for comp_v and B for comp_u) and add missing edges to either component where one has an edge and the other does not, ensuring both matrices become identical.



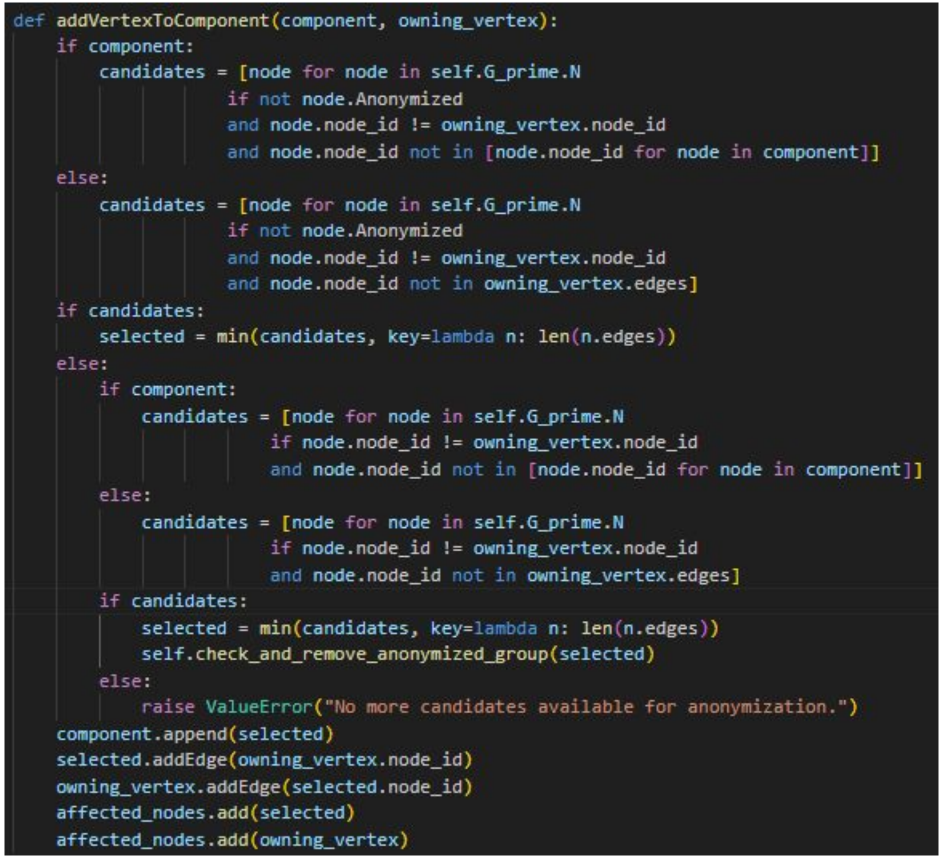
$$\begin{array}{c|cc} & 3 & 2 \\ \hline 3 & 0 & 1 \\ 2 & 1 & 0 \end{array} \;\Bigg|\; \begin{array}{c|cc} & 4 & 2 \\ \hline 4 & 0 & 0 \\ 2 & 0 & 0 \end{array} \longrightarrow \begin{array}{c|cc} & 4 & 2 \\ \hline 4 & 0 & 1 \\ 2 & 1 & 0 \end{array}$$

Vertex addition

A vertices are added during component balancing and select a best candidate:

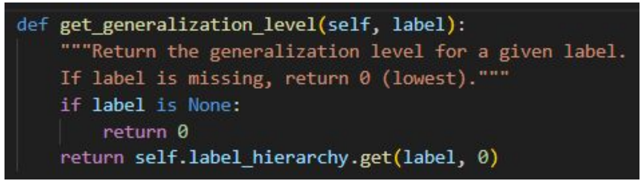


```
while len(comp_v) < len(comp_u):
    addVertexToComponent(comp_v, seed_vertex)
while len(comp_u) < len(comp_v):
    addVertexToComponent(comp_u, candidate_vertex)
```



```
def addVertexToComponent(component, owning_vertex):
    if component:
        candidates = [node for node in self.G_prime.N
                    if not node.Anonymized
                    and node.node_id != owning_vertex.node_id
                    and node.node_id not in [node.node_id for node in component]]
    else:
        candidates = [node for node in self.G_prime.N
                    if not node.Anonymized
                    and node.node_id != owning_vertex.node_id
                    and node.node_id not in owning_vertex.edges]
    if candidates:
        selected = min(candidates, key=lambda n: len(n.edges))
    else:
        if component:
            candidates = [node for node in self.G_prime.N
                        if node.node_id != owning_vertex.node_id
                        and node.node_id not in [node.node_id for node in component]]
        else:
            candidates = [node for node in self.G_prime.N
                        if node.node_id != owning_vertex.node_id
                        and node.node_id not in owning_vertex.edges]
        if candidates:
            selected = min(candidates, key=lambda n: len(n.edges))
            self.check_and_remove_anonymized_group(selected)
        else:
            raise ValueError("No more candidates available for anonymization.")
    component.append(selected)
    selected.addEdge(owning_vertex.node_id)
    owning_vertex.addEdge(selected.node_id)
    affected_nodes.add(selected)
    affected_nodes.add(owning_vertex)
```

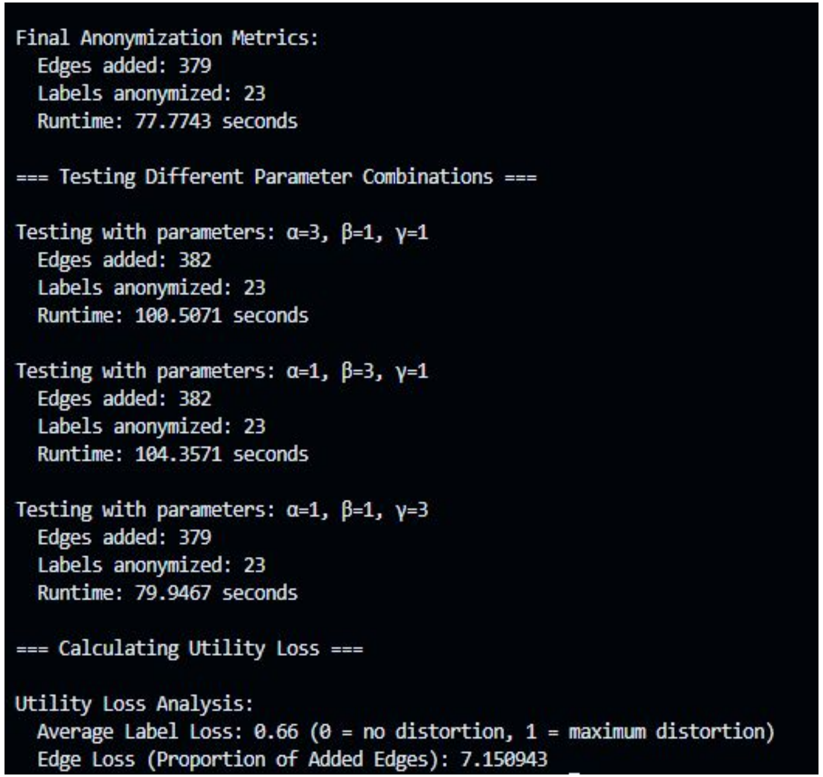
Label Generalization



```
def get_generalization_level(self, label):
    """Return the generalization level for a given label.
    If label is missing, return 0 (lowest)."""
    if label is None:
        return 0
    return self.label_hierarchy.get(label, 0)
```

Empirical Evaluation

- **Computational Complexity:** The algorithm has exponential complexity, making it infeasible to test on large datasets.
- **Experimental Constraints:** Due to time and computational limitations, we conducted experiments on very small datasets.
- **Synthetic Data Generation:** A fake data generator was created to generate datasets of any size, allowing controlled testing.
- **Limitations of Small Datasets:** The algorithm is designed for larger graphs, so testing on small datasets (30 nodes) does not reflect real-world performance. Results may not be reliable or indicative of actual scalability and effectiveness.
- **Experimental Outcome:** Presented anonymization results on a 30-node dataset, highlighting performance issues due to limited data.



```
Final Anonymization Metrics:
  Edges added: 379
  Labels anonymized: 23
  Runtime: 77.7743 seconds

=== Testing Different Parameter Combinations ===

Testing with parameters: α=3, β=1, γ=1
  Edges added: 382
  Labels anonymized: 23
  Runtime: 100.5071 seconds

Testing with parameters: α=1, β=3, γ=1
  Edges added: 382
  Labels anonymized: 23
  Runtime: 104.3571 seconds

Testing with parameters: α=1, β=1, γ=3
  Edges added: 379
  Labels anonymized: 23
  Runtime: 79.9467 seconds

=== Calculating Utility Loss ===

Utility Loss Analysis:
  Average Label Loss: 0.66 (0 = no distortion, 1 = maximum distortion)
  Edge Loss (Proportion of Added Edges): 7.150943
```