CLASS-BASED GRAPH ANONYMIZATION FOR SOCIAL NETWORK DATA

DATA PROTECTION & PRIVACY Project
a.a. 2023/24

Kevin Cattaneo - S4944382
Riccardo Isola – S4943369

Università di Genova

INDEX

- Social network context
- Data Generation
- Algorithm on paper
- Our implementation
- Algorithm complexity
- Privacy level assessment
- Utility level assessment
- Statistical metrics

SOCIAL NETWORK CONTEXT

- Nowadays social networks, such as Facebook and Instagram, have created so far large quantities of data about **interactions** within these networks.

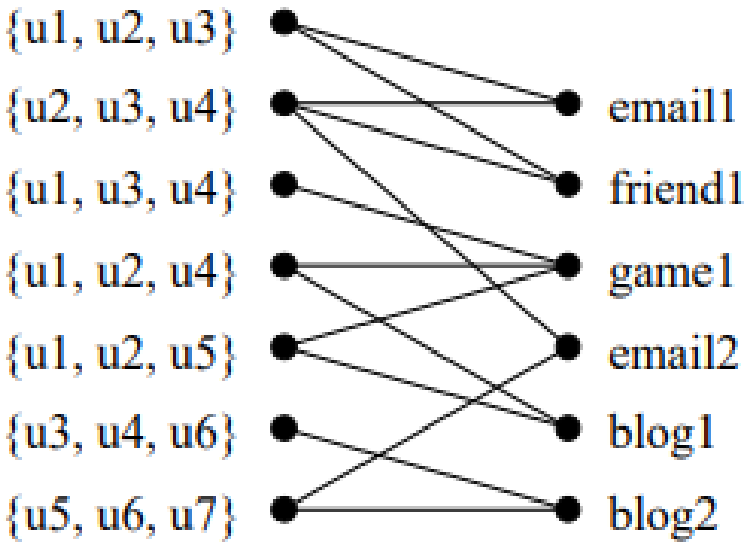
- Such data contains many **private details** about individuals, so anonymization is required both for **privacy of subjects** and to allow **statistical operations** on them

- Many datasets are most naturally represented as **graph structures**, with a variety of types of link connecting sets of entities in the graph.

- An example of this is presented by Online Social Networks (OSNs), which allow users to identify other users as "friends" or "**followers**/ing"

DESCRIPTION OF THE PAPER

- The paper proposes some techniques to anonymize interaction within a graph by using “label lists”
- A label list is a list of possible identifiers of a node, which contains the original node and will substitute it in the anonymized graph
- The choice of the label lists is not trivial, since some of them can be vulnerable to “information leakage”

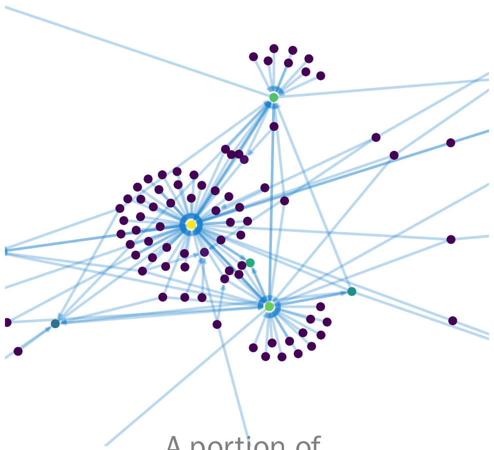


(a) Arbitrary label lists example

Information leakage example by choosing arbitrary label lists

DATA GENERATION

- We generate Italian users with the standard information asked by a social network, having username as unique identifier
- We chose to generate graph data (the following of a user) with **scale_free_graph**, a function from the library **NetworkX**
- The **scale_free_graph** function is typically associated with the Barabási-Albert model for generating scale-free networks, and so we used it to represent a realistic network as much as possible.



A portion of generated graph

```
class User(BaseModel):
    username: ID          # EI
    name: str             # EI
    surname: str          # EI
    birth_date: date      # QI
    gender: Gender        # QI
    cap: int              # QI
    address: str          # QI
    city: str             # QI
    phone_number: str     # QI
    email: str            # EI
```

User node

```
multigraph = scale_free_graph(
    n=len(users),
    alpha=alpha,
    beta=beta,
    gamma=gamma,
    delta_in=delta_in,
    delta_out=delta_out,
    seed=seed,
)
assert len(multigraph) == len(users)
graph = DiGraph(
    {users[id]: [users[i] for i in multigraph[id]] for id in multigraph}
)
graph.remove_edges_from(selfloop_edges(graph))
```

Association User → Following

ALGORITHM ON PAPER

- We partition nodes in classes, that holds a SAFETYCONDITION
- The SAFETYCONDITION prevents the information leakage discussed previously
- The safety comes when any node participates in interactions with at most one node in any class
- The SAFETYCONDITION test can be implemented efficiently by maintaining for each class a list of all nodes which have an interaction with any member of the class
- Classes can have at most *m* elements

Algorithm 1: DIVIDENODES(V,E)

```
SORT(V);
for v ∈ V do
    flag ← true;
    for class c do
        if SAFETYCONDITION(c,v) and SIZE(c) < m then
            INSERT(c,v);
            flag ← false; break;
    if flag then INSERT(CREATENEWCLASS(),v,E);
```

OUR IMPLEMENTATION

- We implement the algorithm preserving the syntax as much as possible and used set operation to perform inserts and checks

- The class holds two sets:
  - The first with the users of that class
  - The second with the following of all users of the class, and so of their interactions

- The SAFETYCONDITION test has been implemented by verifying the intersection between the current class and the list of users of v (plus v, if already there), to check if any of these interaction is present

- Also, our implementation supports directed graphs (paper extension)

```python
def divide_nodes[N](
    V: Graph[N],
    m: int,
    ordering: Callable[[N], Ordering],
) -> list[frozenset[N]]:
    C: list[tuple[set[N], set[N]]] = []

    def safety_condition(c: tuple[set[N], set[N]], v: N) -> bool:
        _, sc = c
        return not bool(sc & {*V[v], v})

    def insert(c: tuple[set[N], set[N]], v: N):
        cls, sc = c
        cls.add(v)
        sc |= {*V[v]}
        sc.add(v)

    def create_new_class():
        c: tuple[set[N], set[N]] = (set(), set())
        C.append(c)
        return c

    for v in sorted(V, key=ordering):
        flag = True
        for c in C:
            if safety_condition(c, v) and len(c[0]) < m:
                insert(c, v)
                flag = False
                break
        if flag:
            insert(create_new_class(), v)
    assert {*V} == {u for c, _ in C for u in c}
    return [frozenset(c) for c, _ in C]
```

ALGORITHM COMPLEXITY - I

- As stated in the paper, the cost of the algorithm is at most: $O(|V||E|\log|V|)$, with V the number of nodes and E the number of edges

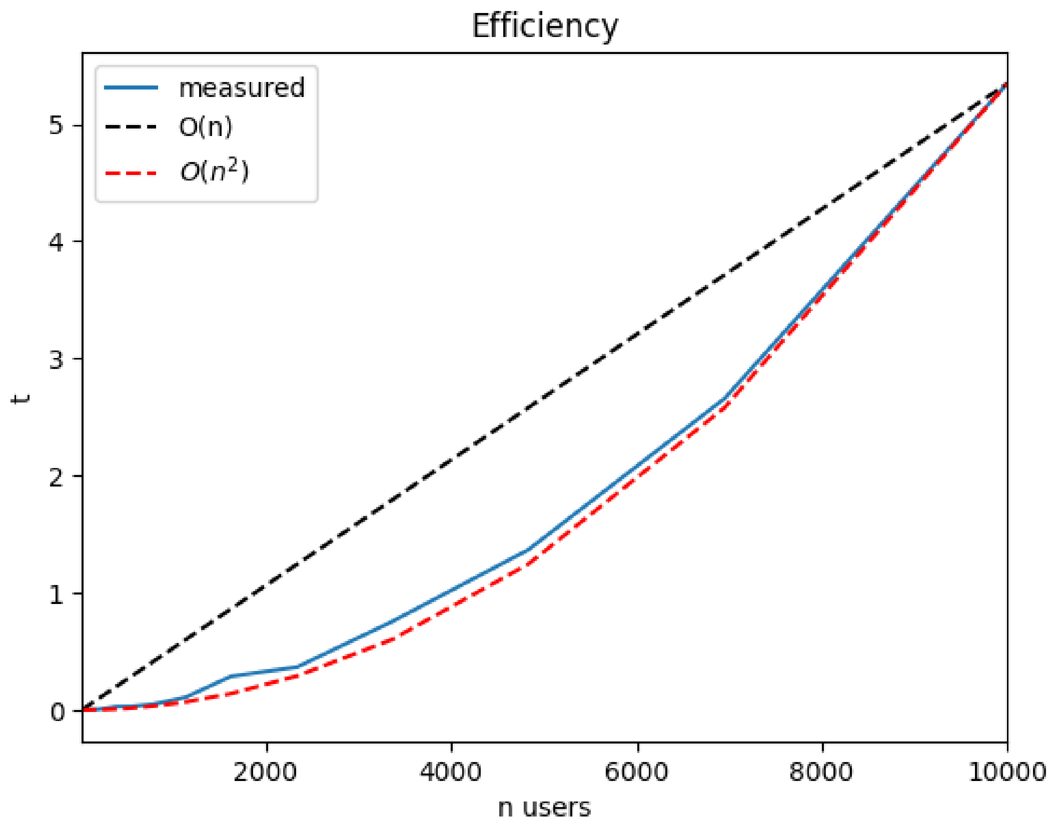
- With our implementation we did *better* (in the average case), since we use hash sets instead of balanced trees: $O(|E||V|) + O(|V|\log|V|)$.

- We obtain the effective difference in term of costs when the number of nodes (and so edges, according to our distribution) grows up to 100 and beyond.

```python
def divide_nodes[N](
    V: Graph[N],
    m: int,
    ordering: Callable[[N], Ordering],
    progress: bool = True,
) -> list[frozenset[N]]:
    # O(|E||V|) + O(|V|log|V|)
    C: list[tuple[set[N], set[N]]] = []

    def safety_condition(c: tuple[set[N], set[N]], v: N) -> bool:  # O(|V[v]|)
        _, sc = c
        return not bool(sc & {*V[v], v})

    def insert(c: tuple[set[N], set[N]], v: N):  # O(|V[v]|)
        cls, sc = c
        cls.add(v)
        sc |= {*V[v]}
        sc.add(v)

    def create_new_class():  # O(1)
        c: tuple[set[N], set[N]] = (set(), set())
        C.append(c)
        return c

    # O(|V|log|V|)
    for v in tqdm(
        sorted(V, key=ordering), desc="creating classes", disable=not progress
    ):  # O(|V|)
        flag = True
        for c in C:  # O(|V|)
            if safety_condition(c, v) and len(c[0]) < m:  # O(|V[v]|)
                insert(c, v)
                flag = False
                break
        if flag:
            insert(create_new_class(), v)
    assert {*V} == {u for c, _ in C for u in c}
    return [frozenset(c) for c, _ in C]  # O(|V|)
```

ALGORITHM COMPLEXITY - II

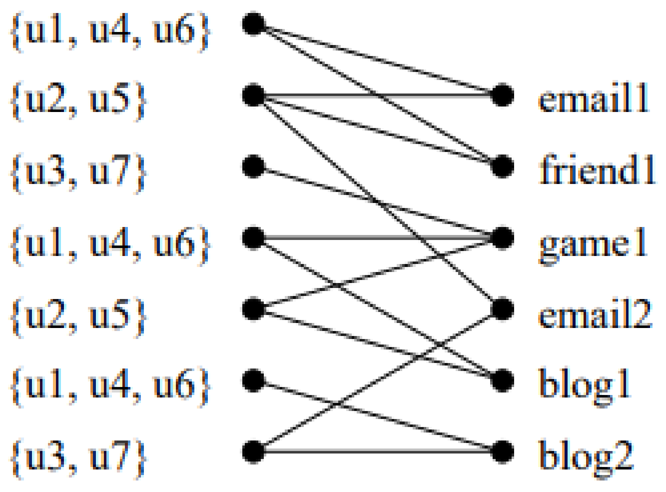
- We perform better than a paraboloid
- But worse than a linear model
- That's because the complexity O(|E||V|) + O(|V|log|V|) depends on the number of edges. The number of edges depends on the data generation, so it is not equal to the number of nodes V



TWO APPROACHES OF ANONYMIZATION

- The paper propose two approaches to anonymize our graph using the generated classes: **uniform lists** or **partitioning.** The latter is more private but the former has a greater utility.

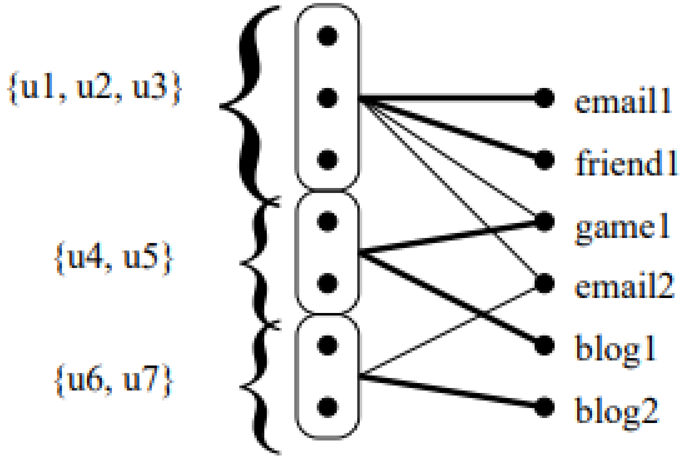
Uniform list



(b) Full pattern example

Edges are not touched

Partition



(c) Partitioning approach (double links in bold)

The graph collapses in a multigraph

SOME DETAILS ON UNIFORM LIST

- The uniform list generation follow the formula:

$$\text{list}(p,i) = \{u_{i+p_0 \mod m}, u_{i+p_1 \mod m}, \cdots u_{i+p_{k-1} \mod m}\}.$$

- *p* is a set of k positive integers

- Also here we have two special cases:
  - **Prefix pattern:** where the pattern *p* is the sequence of numbers from 0 to k-1
  - **Full pattern:** a prefix pattern with k=m

Uniform List Example. Given entities $u_0, u_1, u_2, u_3, u_4, u_5, u_6$ and the pattern $0, 1, 3$, we form label lists to assign to nodes as:

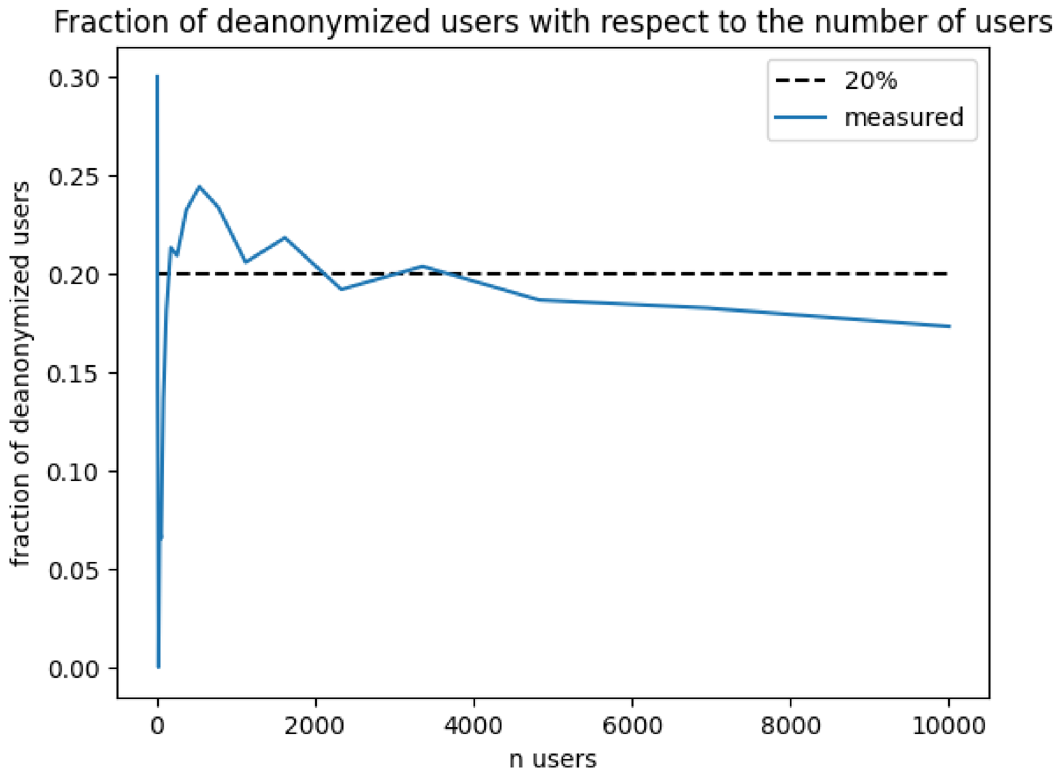
$$\{u_0, u_1, u_3\} \quad \{u_1, u_2, u_4\} \quad \{u_2, u_3, u_5\} \quad \{u_3, u_4, u_6\}$$
$$\{u_4, u_5, u_0\} \quad \{u_5, u_6, u_1\} \quad \{u_6, u_0, u_2\} \quad \square$$

PRIVACY LEVEL ASSESSMENT - I

- We assess our privacy level on our anonymized graph trying to retrieve information about some users

- The main idea is to check how many classes have just one user and so the ones that have not been anonymized, so we can 'de-anonymize' them since they are not mixed with other users in a class

- We observe that in average 20% of users are alone in a class, tested on a sample of 10k users



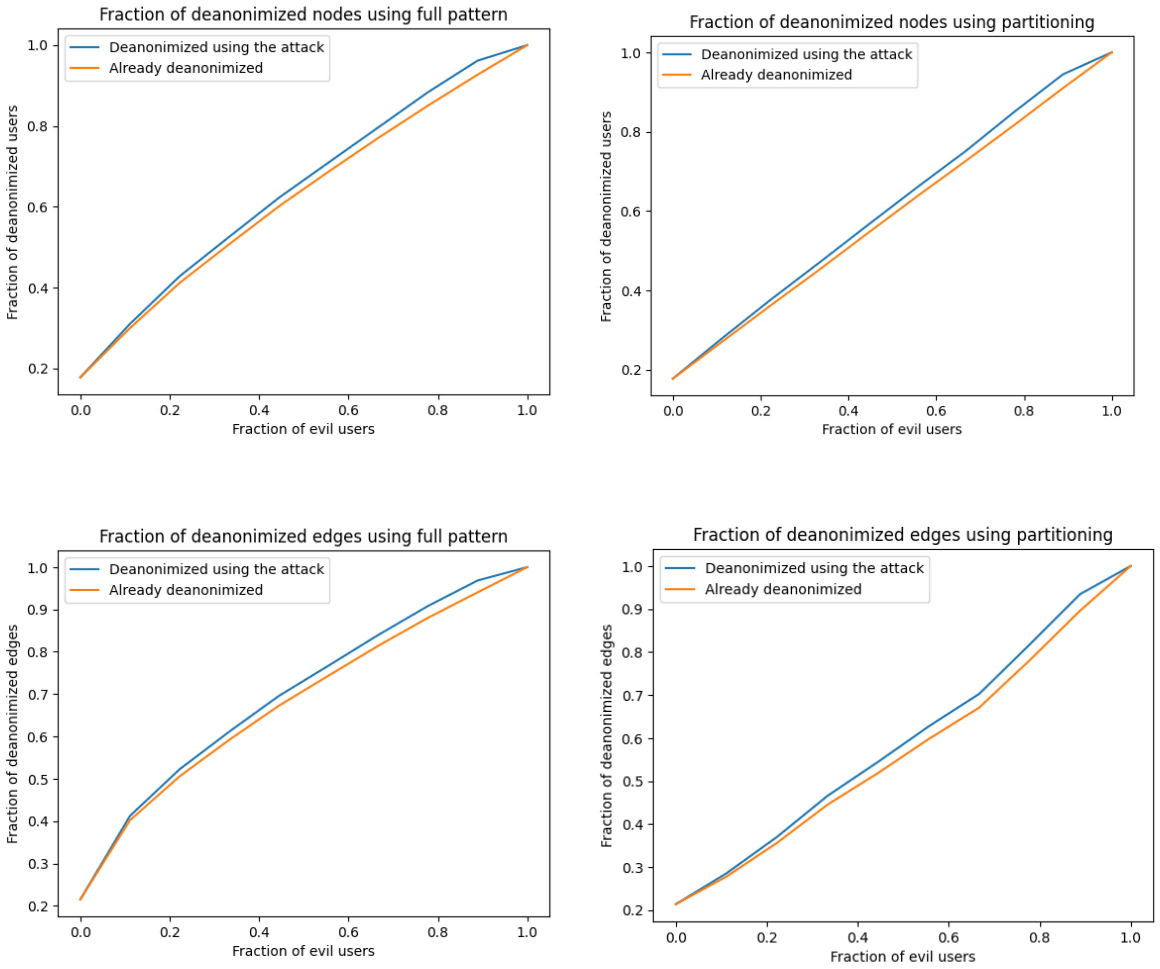
PRIVACY LEVEL ASSESSMENT - II

- We also tried to perform an attack to our graph by creating fake n users and link them to other people as friends

- After anonymization I will discover where my n users have been distributed, with the other non-friend users

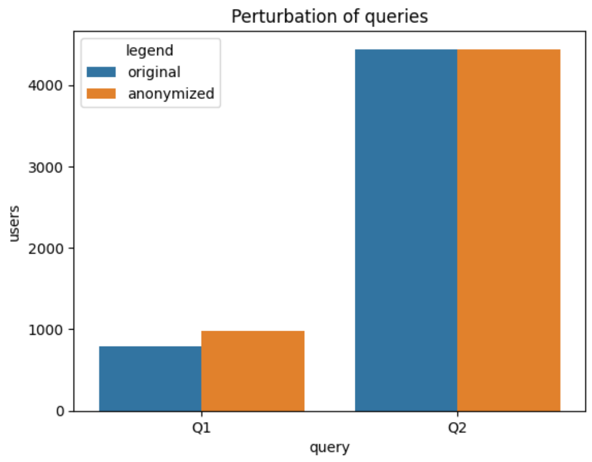
- In this way I can break the anonymization of the nodes in the lists and so I can deanonymize all the edges of these discovered nodes

- Notice that in the partitioning approach the attacker cannot use its friends to deanonymize more nodes



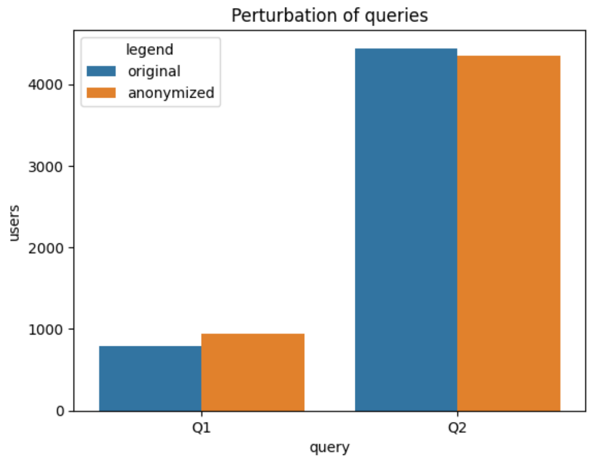
UTILITY LEVEL ASSESSMENT - I

- We assess our utility level on our anonymized graph following the queries suggested by the paper and adapting them to our dataset
- To do that we sample a random graph consistent with the anonymized data
- Q1. «How many people of age > 50 follows younger people of age < 20?»
  - We want to find eventual intergenerational interactions, mentorships, or collaborative engagements between individuals of disparate age groups within the represented network
- Q2. «How many people follow users with in-degree > 10?»
  - We want to find the size of “fan communities” we have created within our network
- In particular we want to observe the perturbation that anonymization process has done quering the two version of the dataset



Uniform list approach

Perturbation of the query Q1 - 23,92%
Perturbation of the query Q2 - 0,00%



Partitioning approach

Perturbation of the query Q1 - 18,73%
Perturbation of the query Q2 - 2,00%

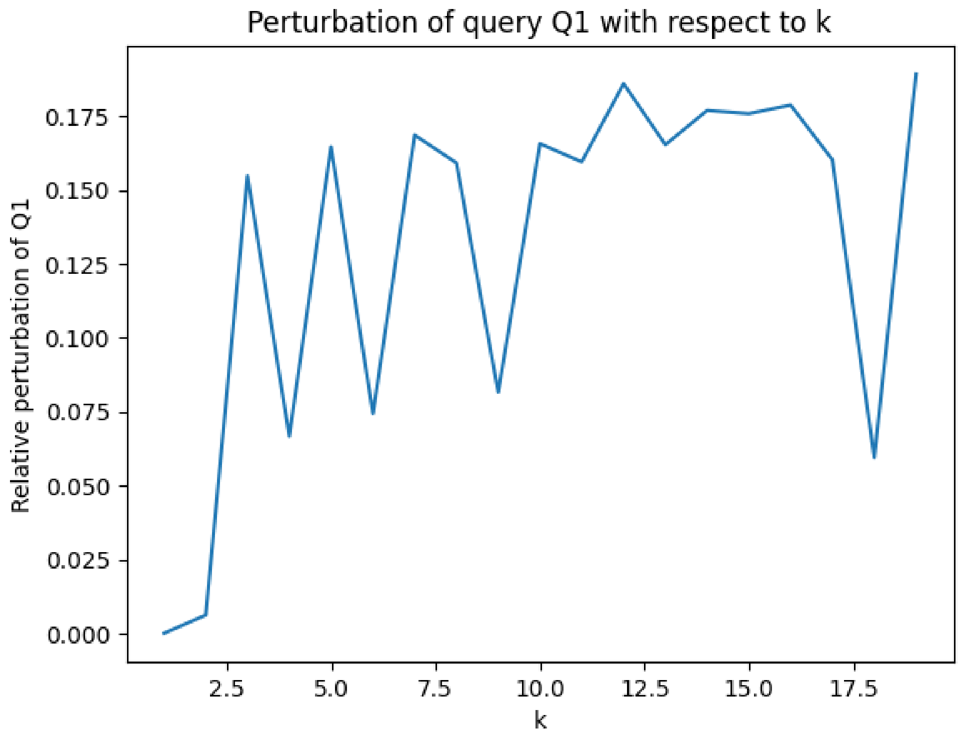
The perturbation involves only the results of queries that lookup to information of both nodes and edges

UTILITY LEVEL ASSESSMENT - II

- Focus on the first query Q1 (intergenerational links)

- We tried to use different *k* values for the indexes chosen from the prefix pattern regarding the uniform list approach

- The perturbation in this example is independent to the value of k, so for this model a high value of k is better since it increase the privacy level without compromising the utility



STATISTICAL INFORMATION

- In the following operations we mark the differences between the two version of the graph, the raw one and the anonymized counterpart

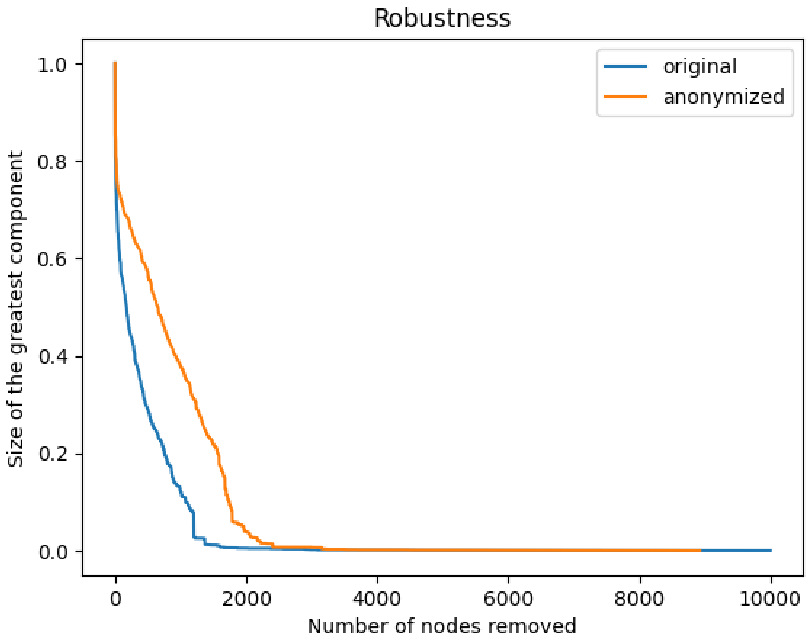
- We focus our statistics on the partitioning approach

- The metrics we choose to compute are:
  - Node removal robustness
  - Diameter and mean degree
  - Closeness and betweenness

NODE REMOVAL ROBUSTNESS

- Measure how well a network maintains its structure and functionality when individual nodes are systematically or randomly removed, without significant distruption

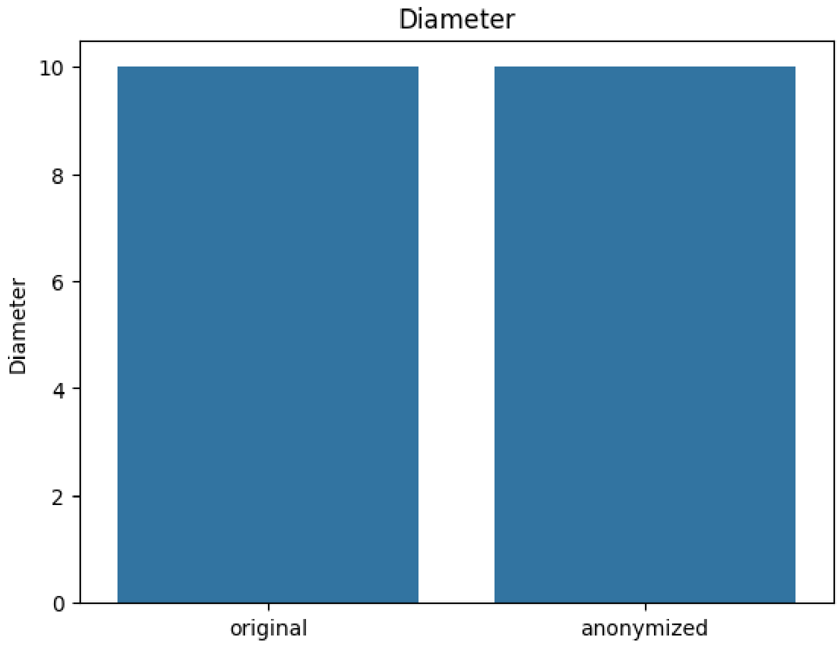
- We expect much more robustness since the partitioning approach shares the interactions with the users of the same class, that bring us to re-distributes edges among the users of the same class when performing the query.



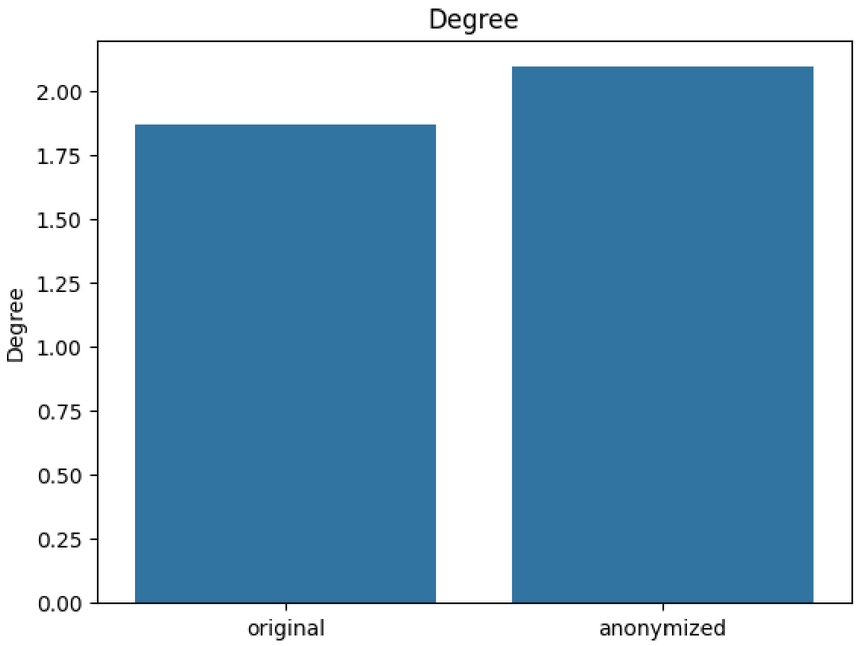
DIAMETER AND MEAN DEGREE

- **Diameter** represents the maximum distance between any pair of nodes in the graph
  Degree is the number of edges (ingoing or outgoing) of a node

- We observe no change on the diameter while an increasing of degree since we have more connections between users of the same class (each user of a class adds to its followings all the others of the users in the same class)



Perturbation - 0,00%

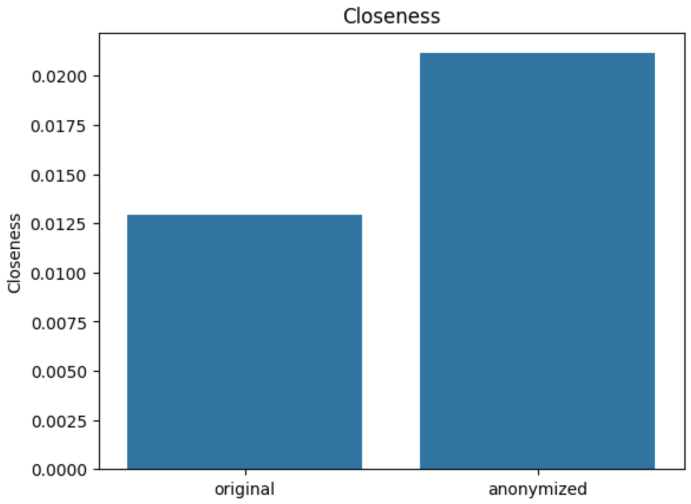


Perturbation – 12,07%

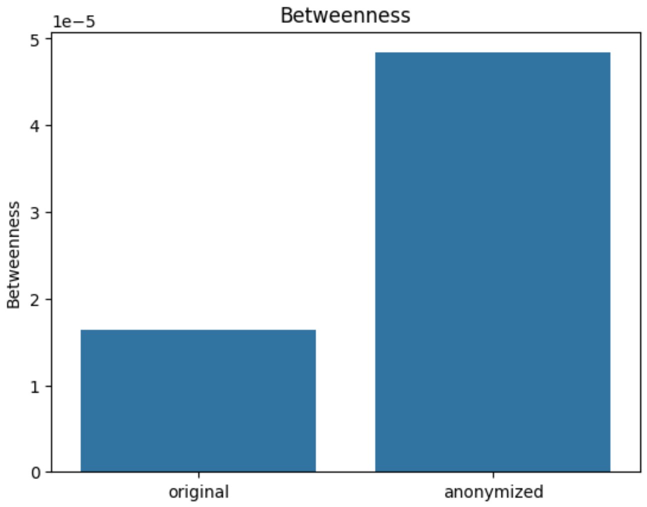
CLOSENESS AND BETWENNESS

- **Closeness** is a measure of how centrally located a node is within the network. It is calculated as the reciprocal of the average shortest path length from a node to all other nodes.
  Betweenness measures the extent to which a node lies on the shortest paths between other nodes in a network. Nodes with high betweenness centrality connect mostly and different parts of the network.

- Since we have more edge distribution, the importance of each nodes is increased, and also the betweenness does by sharing more paths



Perturbation – 63,82%



Perturbation – 194,65%

BIBLIOGRAPHY

- http://upload.wikimedia.org/wikipedia/commons/a/a5/Instagram_icon.png
- http://pngimg.com/uploads/facebook_logos/facebook_logos_PNG19748.png
- https://pngimg.com/uploads/linkedIn/linkedIn_PNG7.png
- https://2023.aulaweb.unige.it/pluginfile.php/314085/mod_page/content/7/5%20-%20Class-based%20graph%20anonymization%20for%20social%20network%20data.pdf

QUESTIONS?