A Workflow for Differentially-Private Graph Synthesis

Davide Proserpio
Boston University
dproserp@cs.bu.edu

Sharon Goldberg
Boston University
goldbe@cs.bu.edu

Frank McSherry
Microsoft Research
mcsherry@microsoft.com

ABSTRACT

We present a new workflow for differentially-private publication of graph topologies. First, we produce differentially-private measurements of interesting graph statistics using our new version of the PINQ programming language, **Weighted PINQ**, which is based on a generalization of differential privacy to weighted sets. Next, we show how to generate graphs that fit *any* set of measured graph statistics, even if they are inconsistent (due to noise), or if they are only indirectly related to actual statistics that we want our synthetic graph to preserve. We combine the answers to Weighted PINQ queries with an incremental evaluator (Markov Chain Monte Carlo (MCMC)) to synthesize graphs where the statistic of interest aligns with that of the protected graph. This paper presents our preliminary results; we show how to cast a few graph statistics (degree distribution, edge multiplicity, joint degree distribution) as queries in Weighted PINQ, and then present experimental results synthesizing graphs generated from answers to these queries.

Categories and Subject Descriptors

H.3 [**Online Information Services**]: Data Sharing

General Terms

Security, Algorithms, Measurement

Keywords

Differential Privacy, Graphs, Privacy, Social Networks

1. INTRODUCTION

Despite recent advances in query languages [8, 12] that support differential-privacy [2], several emerging areas remain underserved by these languages. Perhaps the most notable is social graph analysis, where edges in the graph reflect private information between the nodes. Informally, differential privacy guarantees that the presence or absence of individual records (edges) is hard to infer from the analysis output. Because most interesting social graph analyses reflect paths through the graph, it is difficult to isolate the influence of a single edge, and mask its presence or absence.

1.1 Exploiting non-uniform noise...

To illustrate this difficulty, consider the problem of producing a differentially private measure of a fundamental graph statistic, the *joint degree distribution* (JDD): the counts, for each pair (d_1, d_2), of the number of edges incident on nodes with degree d_1 and degree d_2.

If the maximum node degree in the graph was $d_{\max}$, a naive application of sensitivity-based differential privacy would require that noise proportional to $4d_{\max} + 1$ is added to the count at each JDD entry; this would protect privacy even in the worst case, where $d_1 = d_2 = d_{\max}$. While this is great for privacy, the downside is that when $d_{\max}$ is large, slathering on noise like this ruins the accuracy of our results.

Is it really necessary to add so much noise to all entries of the JDD? Happily, the answer is no. The analysis of [13] shows that the noise required to protect the privacy of the JDD can be *non-uniform*: for each (d_1, d_2) entry of the JDD, it suffices to add noise proportional to $4\max(d_1, d_2)$. Indeed, one consequence of our work is that these sorts of non-uniformities exist in many graph analysis problems, *e.g.,* counting triangles, motifs, *etc.*. In each case, features on low degree vertices are measured very accurately, with less accurate measurements on higher degree vertices.

1.2 ...without custom technical analyses!

We can significantly improve the accuracy of differentially private graph measurements by exploiting opportunities to apply noise non-uniformly [13]. However, each new measurement algorithm requires a new non-trivial privacy analysis, that can be quite subtle and error prone. Moreover, existing languages like PINQ are no help, as they are explicitly designed not to rely on custom analyses by unreliable users.

In this work, we use a new declarative programming language, **Weighted PINQ**, designed to allow the user to exploit opportunities for non-uniformity while still automatically imposing differential privacy guarantees. The key idea built into the language is the following observation: rather than add noise non-uniformly to various aggregates of integral records, we add noise uniformly to aggregates of records whose weights have been *non-uniformly scaled down*. The operators in the language are perfectly positioned to identify problem records and scale down their weight, rather than poison the analysis by increasing the noise for all records.

WOSN'12, August 17, 2012, Helsinki, Finland.

1.3 A new workflow for graph release.

We describe a three-phase workflow that can generate *synthetic graphs* that match any properties of the true (secret) graph that we can measure in Weighted PINQ:

Phase 1. Measure the secret graph. First, we cast various graph properties as queries in Weighted PINQ, and produce differentially private (noisy) measurements of the secret graph. Some of the queries will have natural interpretations, *e.g.,* a degree complementary cumulative distribution function (CCDF) query (Section 3.1). Other queries will be only indirectly meaningful, in that they constrain the set of graphs which could have led to them but they will not explicitly reveal the quantity of interest (Section 5.4).

At the end of Phase 1, we discard the secret graph and proceed using only the differentially private measurements taken from it. When these measurements are sufficient, we can report them and stop. However, we can go much further using *probabilistic inference* [14], *i.e.,* by fitting a random graph to our measurements. While our measurements are noisy, they constrain the set of plausible graphs that could lead to them. Moreover, the properties of this set of plausible graphs may be very concentrated, even if we did not measure these properties directly. For example, the set of graphs fitting both our noisy JDD measurements and our (very accurate) degree CCDF measurements should have assortativity very close to the secret graph, even if they have little else in common. Thus, we proceed as follows:

Phase 2. Create a "seed" synthetic graph. Our measurements typically contain queries about degrees that, cleaned up, are sufficient to seed a simple random graph generator. We do so, as a very primitive approximation to the sort of graph we would like to release.

Phase 3. Correct the synthetic graph. Starting from the seed synthetic graph produced in the second phase, we search for synthetic graphs whose accurate (noise-free) answers to our Weighted PINQ queries are similar to the noisy measurements we took in the first phase. We perform this search with Markov-Chain Monte Carlo (MCMC), a traditional approach from machine learning used to search for datasets matching probabilistic observations, in our case graphs matching noisy answers to weighted PINQ queries.

Generally, MCMC involves proposing a new candidate (synthetic graph) at each iteration, and re-evaluating the Weighted PINQ queries on this graph to see if the fit has improved. While this could be time consuming, each MCMC iteration is designed to introduce only small updates to the candidate graph (*e.g.,* swapping one edge for another), and our implementation can efficiently process small changes to its input [10]. Depending on query complexity, we can process up to tens of thousands of candidate graphs per second.

1.4 Our results and roadmap.

This short paper reports on our initial experiences using our new workflow, with full details in our technical report [10]. We focus on a small number of interesting graph statistics – namely, the degree distribution, edge multiplicity, assortativity, and JDD – and show how our workflow can generate synthetic graphs that match these statistics. But these statistics are by no means the end of the story; we have developed additional queries measuring other statistics of interest, *e.g.,* clustering coefficient, triangles, graph motifs, *etc.*, and are evaluating their efficacy.

We start in Section 2 with a brief overview of Weighted PINQ. In Section 3 we present Weighted PINQ queries for first-order graph statistics, and experimental results of seed synthetic graphs that were produced using only these statistics (*i.e.,* in Phase 2). In Section 4, we show how MCMC can be used to "correct" edge multiplicities in these synthetic graphs, and in Section 5 we show how we used an indirect measurement of the JDD to "correct" the assortativity of our synthetic graphs.

2. WEIGHTED DP AND PINQ

We model a dataset A as a weighting of the records: $A : D \to \mathbb{R}$, where $A(r)$ represents the weight of r in A. Traditionally this number would be a non-negative integer, but we allow any real number. We provide the following generalization of differential privacy:

DEFINITION 2.1. *A randomized computation M provides ϵ-*differential privacy *if for any weighted datasets A and B, and any set of possible outputs $S \subseteq Range(M)$,*

$$\Pr[M(A) \in S] \quad \leq \quad \Pr[M(B) \in S] \times \exp(\epsilon \times \|A - B\|) \; .$$

where $\|A - B\| \;=\; \sum_x |A(x) - B(x)|$.

This definition is equivalent to differential privacy on integral weighted datasets, but is stricter on weighted datasets.

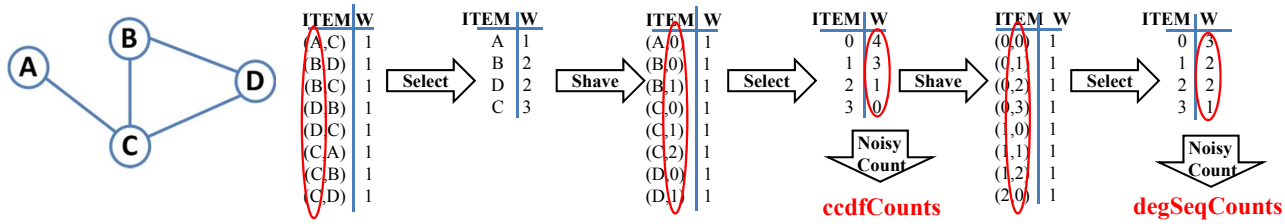
Node vs. edge privacy. Our secret datasets contain directed edges with weight 1.0, where differential privacy masks the presence or absence of each edge. The set of *all* outgoing edges from a vertex v can be protected by weighting down each directed edge by $0.5/d_v$, trading some accuracy for a stronger notion of privacy. Note that this is not as strong as "node privacy", where the existence of each vertex is masked; Ashton Kutcher's *existence* on Twitter would not be masked, though each of his followers' shame would be.

2.1 Weighted PINQ.

Like Privacy Integrated Queries (PINQ) [8], Weighted PINQ is a declarative programming language over datasets that guarantees differential privacy for every program written in the language. We refer the reader to [8] for details on the design philosophy behind these languages, and to [10] for full technical details on Weighted PINQ's operators. Here we only provide a short overview of how queries are written in Weighted PINQ.

Overview. Weighted PINQ can apply two types of operators to a (secret) dataset: transformations, and noisy aggregations. Transformation operators (*e.g.,* `Select`, `Where`, `GroupBy`, `SelectMany`, `Join`, ...) transform a weighted secret dataset while automatically rescaling the resulting record weights. The datasets that result from these transformations remain secret; before they can be exposed to the user, they must be fed to noisy aggregation operators (*e.g.,* `NoisyCount`) that aggregate the weighted secret records, add (ϵ-amplitude Laplace) noise, and expose the results. Each transformation operator scales down record weights in a manner that guarantees differential privacy after noisy aggregation (*e.g.,* `Join` always rescales weights per Equation (1) in Section 5.2).

Weighted PINQ vs. PINQ. Weighted PINQ inherits many of PINQs operators. However, because it acts on records of arbitrary weights (rather than integral weights), it deviates from PINQ in a few important ways:



First, transformations in PINQ that required either scaling up the noise or the privacy parameter ϵ, now scale *down* the weights associated with records. The most germane operators for this note are `SelectMany` (produces many records (*e.g.,* k), each with weight scaled down by a factor of k), `GroupBy` (collects records, and results in a group with weight divided by 2), and `Join` (produces the cross-product of records, with weights rescaled as in Equation (1)). `Join` is the workhorse of non-uniformity in our graph analysis.

Second, there is a transformation operator to manipulate weights, `Shave`, which takes a sequence of weights $\{w_i\}$ and transforms each record x with weight w into the set of records $(0, x), (1, x), \ldots$ with weights $w_0, w_1, \ldots$, for as many terms as $\sum_i w_i \leq w$. The functional inverse of `Shave` is `Select`, which can transform each w_i-weighted indexed pair from (i, x) to x whose weight re-accumulates to $\sum_i w_i = w$.

Third, Weighted PINQ's `NoisyCount` now returns a dictionary from records to noised weights, rather than a single noisy count. If one looks up the value of a record not in the input, a weight of zero is introduced and noise added. This generalizes PINQ's `NoisyCount` to weights and multi-output "histogram queries" [2]; to reproduce PINQ's `NoisyCount` we can first map all records to some known value, *e.g.,* `true`.

Writing "good" queries. We emphasize that the fact that a query is expressed using Weighted PINQ operators (*i.e.,* transformations, followed by aggregations) suffices to prove that it provides differential privacy; the task of a user then becomes to write a "good" Weighted PINQ query. A query's quality is judged by (1) its computational complexity and, since results in Weighted PINQ are always noisy, (2) its accuracy. Writing "good" queries requires ingenuity; here we present some example queries that provide high accuracy and performance with little loss of privacy. We note that it can be more challenging to write "good" queries that *directly* measure properties with high *sensitivity* [2] (*e.g.,* graph diameter); one way to get around this could combine *indirect* measurements with probabilistic inference (Section 1.3).

3. CREATING THE SEED GRAPH

We start by creating a "seed" synthetic graph based on queries related to the degree distribution.

Weighted PINQ Operators. The queries in this section use the `NoisyCount` aggregation described in Section 2.1, as well as `Select` and `Where` transformations that function in the same manner as their namesakes in LINQ and PINQ *without* rescaling record weights [8, 10].

3.1 Degree CCDF

We present a Weighted PINQ query for computing the complementary cumulative distribution function (CCDF) of node outdegree. Starting from `edges`, the secret dataset of unit-weight directed edges (n_o, n_i), do:

```
var deqCCDF = edges.Select(edge => edge.src)
                   .Shave(1.0)
                   .Select((index, srcname) => index);

var ccdfCounts  = degCCDF.NoisyCount(epsilon);
```

The steps of the query are depicted in the figure above. We start by transforming the dataset so that each record is a node's name n_j, weighted its outdegree d_j. Next, we shave each n_j record into d_j unique, unit-weight pairs: $(n_j, 0), (n_j, 1)$, ..., $(n_j, d_j - 1)$. By keeping only the index of the pair, we obtain records $i = 0, 1, 2, ..., d_{\max} - 1$, each weighted by the number of nodes in the graph with degree greater than i. Finally, taking a noisy sum the weight of each record gives the outdegree CCDF. (Unsurprisingly, replacing the first line of our algorithm with `edge.Select(edge => edge.tgt)` would result in the indegree CCDF.) Neither `Shave` nor `Select` scale down record's weight; it follows that our CCDF query provides ϵ-differential privacy while preserving all of the weight in our `edges` dataset.

3.2 Degree Sequence

The degree CCDF is the functional inverse of the *degree sequence*, *i.e.,* the monotonically non-increasing sequence of node degrees in the graph, *i.e.,* $d_1, d_2, ..., d_n$ such that $d_i \geq d_{i+1}$. To get the degree sequence, we need only *transpose* the x- and y-axis of a plot of the degree CCDF. We can do this in weighted PINQ without scaling down the weight of any of our records:

```
var deqSeq = degCCDF.Shave(1.0)
                    .Select((index, degree) => index);

var degSeqCounts  = degSeq.NoisyCount(epsilon);
```

This query, illustrated in the figure above, is actually a Weighted PINQ implementation of an ϵ-differential privacy algorithm proposed by Hay *et al.* [3]!

Query complexity. At every point in these two queries we have at most $|E|$ items, resulting in a storage complexity of $|E|$. Each transformation takes linear time, so running these queries on a new (protected) graph takes time $O(|E|)$.

3.3 Relating the degree sequence & CCDF

Hay *et al.* observed that a significant amount of noise can be "cleaned up" in the degree sequence by using isotonic regression (because the degree sequence is known to be non-increasing) [3, 6]. We observe that the same is true for the CCDF, and moreover the degree sequence and CCDF give accurate information about different aspects of the graph: the former accurately reports the graph's highest degrees, whereas the latter (its transpose) better reports the numbers of low degree nodes. While PAVA [6], and independently [3], can regress either the CCDF or the degree sequence to a consistent non-increasing sequence, in [10] we

develop a regression technique based on shortest paths that finds a single sequence optimizing the pair of measurements, producing a degree sequence that is accurate for both the high- and low-degree nodes.

	Nodes	Edges	Assort'y	max iDeg	max oDeg
AS Graph	14233	32600	-0.306	172	2389
Collab.	5242	28980	+0.659	81	81

Table 1: Original (secret) graph statistics

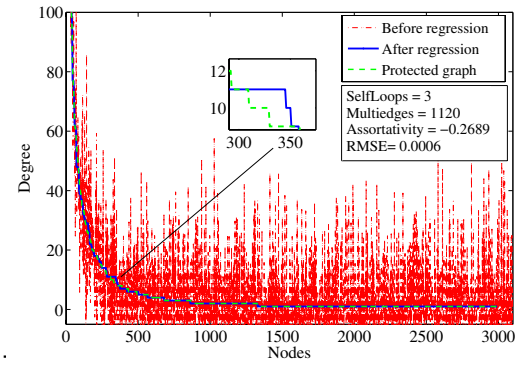


Figure 1: Degree Sequence, ARIN AS graph.

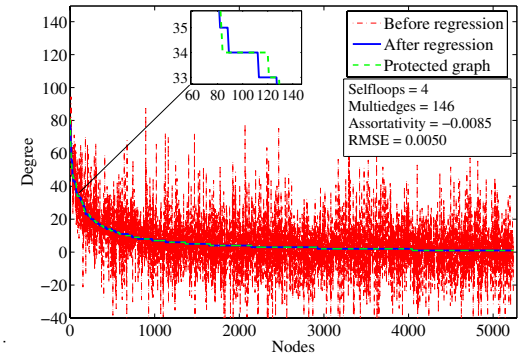


Figure 2: Degree Sequence, Collab. Graph.

3.4 Results: Initial synthetic graphs.

We used a version of the 1K-graph generator to generate a random graph that fits (1) our "cleaned-up" non-increasing outdegree sequence (that has privacy cost 2ϵ since it was generated from two ϵ-dp measurements, namely, degree sequence and CCDF) and (2) our "cleaned-up" indegree sequence as well as (3) an ϵ-dp measure of the number of nodes in the graph (see query in [10]). Setting $\epsilon = 0.1$, these synthetic graphs use a total privacy cost of $5\epsilon = 0.5$.

Due to space limitations, we only present results for two graphs; a graph of autonomous systems in the ARIN region [1] (available on our project website), the Arxiv GR-QC collaboration graph [7]. Statistics about the original graphs are in Table 1. In Figures 2, 1, we plot measured degree sequences, both before regression and after regression, and compare them to the degree sequence of the actual protected graph. Regression removes most of the noise in the measured degree sequence; the normalized root-mean-square-error (RSME) after regression for each graph is $< 1\%$.

4. MULTIEDGES AND SELF-LOOPS

Figures 2 and 1 reveal that our seed graphs contain self-loops (*i.e.,* edges from a node to itself) and a large number of multiedges (*i.e.,* repeated edges). By writing a Weighted PINQ query to measure the number of multiedges and self-loops, we can cause MCMC to prefer graphs that respect their presence or absence.

Query. The following query uses `Shave` to report the multiplicity of each edge, and then `Select` to classify each as a self-loop or not. The numbers of each type are then counted, with noise. Since we only use `Select` and `Shave`, this query is ϵ-dp without scaling record weights:

```
var multi = edges.Shave(1.0)
                 .Select((i, e) => new
                      Pair(i, e.src == e.tgt ? 1 : 0))
                 .NoisyCount(epsilon);
```

The query is illustrated in the figure above.

MCMC. We fed our seed synthetic graph into MCMC, and let it correct the self-loops and multiplicity. Our edge-swapping MCMC algorithm first repeats all the measurements taken on the protected graphs on the seed synthetic graph; then, at each iteration, it (1) randomly chooses a pair of edges $(n_1, n_2), (n_1', n_2')$ from the synthetic graph, (2) replaces them with $(n_1, n_2'), (n_1', n_2)$, incrementally updates its measurements (which can be done in constant time), and (3) probabilistically decides to either accept the replacement edges, or revert to the original pair of edges [10]. MCMC was consistently able to remove all of the extra multiedges and self loops in the seed graphs depicted in Figure 1 and 2. Because our edge-swapping MCMC algorithm does not alter the in- and out-degree distributions, it only improves the fit to the measurements by correcting self-loop and multi-edges.

Privacy. An $\epsilon = 0.1$ query is added to those used to generate the seed graph, increasing privacy cost to 6ϵ.

5. JOINT DEGREE DISTRIBUTION

Our next objective is to correct the *assortativity* of our synthetic graph. The assortativity can be computed directly from the joint degree distribution (JDD), and gives a measure of degree correlations. Assortativity is high (close to +1) if nodes tend to be connected to nodes of similar degree, low (close to -1) if the opposite is true, and ≈ 0 in a random graph. Fortunately, we need not measure the JDD *directly.* Instead, we measure a property that is indirectly related to the JDD, thus forcing MCMC to fit the synthetic graph to the assortativity of the original graph.

5.1 Using the `GroupBy` transformation

Our JDD query uses the `GroupBy` transformation, which we describe in detail in [10]. For our purposes, on integral datasets this transformation takes a function mapping records to key values and a function from a set of records to

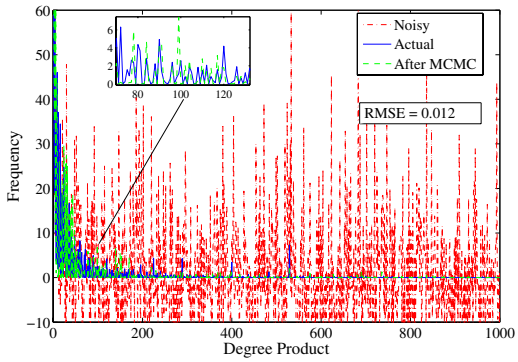


Figure 3: JDD of Collab., 1.0-differential privacy

a result value, and for each observed key emits a pair with weight 0.5 containing the key and the function applied to the corresponding set of records.

We can obtain a dataset containing (node, indegree) pairs, each of weight 0.5, by taking

```
var iDegs = edges.GroupBy(e => e.tgt, l => l.Count());
```

Importantly, these counts are taken without noise. The results of `GroupBy` are still protected data, and may only be examined through noisy aggregation.

5.2 Using the `Join` transformation

Next, we show how to use the `Join` transformation, allowing us to scale down record weights in interesting, non-uniform ways. `Join` takes two datasets, two key selector functions, and a reducer from pairs of elements to a result type. For each pair of records with matching keys, it applies the reduction function and emits the result. In PINQ, `Join` mangled the results when matches were not unique, to prevent multiple release of a single input. Weighted PINQ will release all the records, but with weights scaled down.

Suppose we `Join` two datasets A, B, and let A_k and B_k be the restrictions of A and B, respectively, to those records mapping to a key k under their key functions. For every k and for every pair $(\alpha, \beta) \in A_k \times B_k$, the `Join` operator emits the record $reducer(\alpha, \beta)$ with weight

$$\frac{A(\alpha) \times B(\beta)}{\|A_k\| + \|B_k\|} \tag{1}$$

The weight of each output record is the product of the corresponding weights, divided by the sum of all weights with the same key. This setting of weights ensure that any ϵ-dp measurement of the output provides ϵ-dp for each input. Note that if the same dataset is used twice, in each input for example, it incurs the ϵ cost twice.

As an example, consider applying the `Join` to our `edges` dataset and the `iDegs` dataset produced above:

```
var iDegEdges =
  edges.Join(iDegs, edge => edge.tgt, ideg => ideg.node,
             (edge, ideg) => new Pair(edge, ideg.deg))
```

If the $d_i(v)$ in-neighbors of v are $u_1, \ldots u_{d_i(v)}$, then the restrictions of `edges` and `iDegs` to key v are:

$$\texttt{edges}_v = \{(\text{“}(u_1, v)\text{”}, 1), \ldots, (\text{“}(u_{d_i(v)}, v)\text{”}, 1)\}$$
$$\texttt{iDegs}_v = \{(\text{“}(v, d_i(v))\text{”}, 0.5)\}$$

The outputs for each key v are therefore

$$\text{“}((u_1, v), d_i(v))\text{”}, \text{“}((u_2, v), d_i(v))\text{”}, ..., \text{“}((u_{d_i}, v), d_i(v))\text{”} .$$

We have $\|\texttt{edges}_v\| = d_i(v)$ and $\|\texttt{iDegs}_v\| = 0.5$, so the the weight of every output element equals

$$0.5/(d_i(v) + 0.5) = 1/(2d_i(v) + 1) .$$

Note that the result uses `edges` twice (`iDegs` derives from `edges`) so using the dataset will cost twice the ϵ.

5.3 A direct measure of the JDD.

Given the (edge, indegree) dataset `iDegEdges` described above, we form the analogous (edge, outdegree) dataset `oDegEdges`, and join the two together to obtain the (outdegree, indegree) dataset required for the JDD:

```
var jdd =
  oDegEdges.Join(iDegEdges, o => o.edge, i => i.edge,
                 (i, o) => new Pair(o.deg, i.deg));

var jddCount = jdd.NoisyCount(epsilon);
```

For each edge (u, v), the weighted sets $\texttt{oDegEdges}_{(u,v)}$ and $\texttt{iDegEdges}_{(u,v)}$ both contain single elements, with weights $(2d_o(u) + 1)^{-1}$ and $(2d_i(v) + 1)^{-1}$, respectively. The resulting output record, $(d_o(u), d_i(v))$, has weight $(2d_o(u) + 2d_i(v) + 2)^{-1}$. Taking a `NoisyCount` of the `jdd` dataset provides a 4ϵ-differential privacy measure of the JDD (as each input uses `edges` twice). This corresponds to an ϵ-differential privacy measurement of the true count of (d_1, d_2) edges with noise $8(d_1 + d_2 + 1)/\epsilon$, similar to (but worse than) the $4\max(d_1, d_2)/\epsilon$ from the custom analysis of [13].

5.4 An indirect measure of the JDD.

Noise of magnitude $8(d_1 + d_2 + 1)/\epsilon$ in counts that are often zero and rarely much more can be a serious problem; even $4\max(d_1, d_2)$ was found to be too much to add to all entries in [13]. At this point, the combination of Weighted PINQ and MCMC shines. Instead of directly computing a `NoisyCount` on the `jdd` dataset, we significantly improve our signal-to-noise ratio by bucketing degrees based on information from our degree distribution measurements, and then taking a `NoisyCount` on the larger weight in each bucket:

```
var jddBucketCount = jdd.Select(x => bucket(x, buckets))
                        .NoisyCount(epsilon);
```

The `bucket` function takes the pair $(d_o(u), d_i(v))$ to a pair (x, y) corresponding to the indices of the buckets the degrees land in. Weighted PINQ guarantees differential privacy (we don't need a new analysis) and MCMC focuses on graphs with the same bucket counts (we don't need any new post-processing). All that we need to do is see how well it works.

Query complexity. We have at most $O(|E|)$ items at every point in our JDD queries, resulting in a storage and computational complexity of $O(|E|)$ when we run the query on a new (protected) graph. Incrementally updating the measurement during an iteration of MCMC that swaps edges $(u, v), (u', v')$ takes time $O(d_o(u) + d_i(u') + d_o(v) + d_i(v'))$.

5.5 Results: Fitting the assortativity.

The assortativity of our seed AS graph (Figure 1) is already quite close to that of the original AS graph (Table 1). Therefore, we only present results for the collaboration graph,

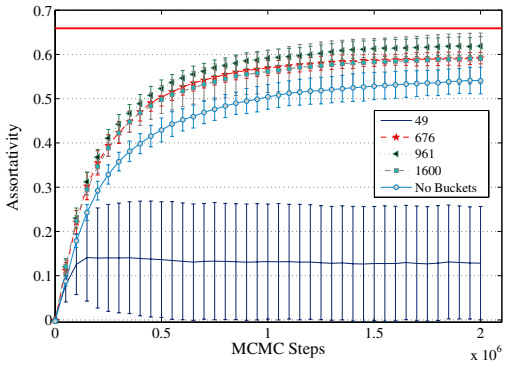


Figure 4: Assort'y vs MCMC steps (Collab.)

generated using the six previously-discussed queries of privacy cost ϵ, as well as a JDD query with cost 4ϵ, for total privacy cost of $10\epsilon = 1.0$.

Figure 3. We plot the "scaled-down JDD" where the count in each pair (d_o, d_i) is scaled by $(2d_o + 2d_i + 2)^{-1}$. We compare the measured value (`jddCount`) to that of the original graph, and the synthetic graph after 2M iterations of MCMC. (For simplicity, we plot the degree product $d_o d_i$ on the x-axis instead of the two-dimensional degree pairs (d_o, d_i)). While our measured JDD is extremely noisy for the high degree pairs, it has much better accuracy for the low degree pairs. If we use MCMC to combine the noisy JDD measurement with our highly-accurate degree distribution measurements (Figure 2), we clean up most of this noise; our scaled-down JDD has normalized RSME of only 0.01 (averaging five trials) after MCMC.

Figure 4. Next, we consider the effect of bucketizing. We plot the assortativity of our synthetic graph versus the number of iterations of MCMC, for different choices of buckets. We plot the mean of five experiments and their relative standard deviation. Our synthetic graphs start out with assortativity ≈ 0, and climb towards the target assortativity of 0.65 as MCMC proceeds. Moreover, there seems to be a 'happy-medium' for bucketizing; with too few buckets, we lose too much information, and with too many buckets (or none at all), our signal-to-noise ratio is too low. For this configuration, using $31^2 = 961$ buckets works best; averaging five trials we obtain assortativity 0.62, and normalized RSME of 0.003 and 0.009 in the degree CCDF and "scaled-down JDD" respectively, with an average of 6 spurious multiedges and 10 self-loops (the real collaboration graph has 12 self loops). Note that different graphs or choices of the privacy parameter ϵ may require different bucketizing strategies; we are characterizing this as part of our ongoing work.

6. RELATED WORK

Since the introduction of differential privacy [2], numerous bespoke analyses have emerged for specific problems, as well as general tools such as PINQ [8]. The bulk of this work has focused on statistical analyses of tabular data. Although graph-structure data can be viewed in a tabular form (each edge has a "source" and "destination" attribute), graph queries typically result in many `Join` operations over the tables, requiring excessive amounts of additive noise using standard tools [11]. Bespoke analyses have recently emerged for degree distributions [3], joint degree distribution (and assortativity) [13], triangle counting [9], and some generalizations of triangles [5]. We can provide analogues of each of these approaches in Weighted PINQ, typically matching proven bounds (within constants) and always exploiting the non-uniformities described in Section 1.1.

Many graph analyses satisfy privacy definitions other than differential privacy [4]. These definitions generally do not exhibit the robustness of differential privacy, and a comparison is beyond the scope of this note.

7. FUTURE WORK

In this short paper, we provided an overview of our new workflow for differentially-private graph synthesis, and presented initial results of our experiments to generate synthetic graphs that fit the degree distribution, edge multiplicity, and assortativity of the protected graph. But this is only a first step; our current work involves developing improved queries of these graph statistics, and expanding to new ones, including triangles, clustering coefficient, motifs, and more.

8. REFERENCES

[1] Ying-Ju Chi, Ricardo Oliveira, and Lixia Zhang. Cyclops: The Internet AS-level observatory. *ACM SIGCOMM CCR*, 2008.

[2] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography*, volume 3876 of *Lecture Notes in Computer Science*, pages 265–284. 2006.

[3] M. Hay, Chao Li, G. Miklau, and D. Jensen. Accurate estimation of the degree distribution of private networks. In *IEEE ICDM '09*, pages 169 –178, dec. 2009.

[4] Michael Hay, Kun Liu, Gerome Miklau, Jian Pei, and Evimaria Terzi. Tutorial on privacy-aware data management in information networks. Proc. SIGMOD'11, 2011.

[5] Vishesh Karwa, Sofya Raskhodnikova, Adam Smith, and Grigory Yaroslavtsev. Private analysis of graph structure. In *Proc. VLDB'11*, pages 1146–1157, 2011.

[6] Jan De Leeuw, K. Hornik, and P. Mair. Isotone optimization in r: Pool-adjacent-violators algorithm (pava) and active set methods. *Journal of statistical software*, 32(5), 2009.

[7] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. KDD*, 1(1), 2007.

[8] Frank D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proc. SIGMOD '09*, pages 19–30, 2009.

[9] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Smooth sensitivity and sampling in private data analysis. In *ACM STOC '07*, pages 75–84, 2007.

[10] Davide Proserpio, Sharon Goldberg, and Frank McSherry. A workflow for differentially-private graph synthesis. Technical report, March 2012.

[11] Vibhor Rastogi, Michael Hay, Gerome Miklau, and Dan Suciu. Relationship privacy: output perturbation for queries with joins. In *Proc. PODS '09*, 2009.

[12] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: security and privacy for mapreduce. In *Proc. USENIX NSDI'10*, pages 20–20. USENIX Association, 2010.

[13] Alessandra Sala, Xiaohan Zhao, Christo Wilson, Haitao Zheng, and Ben Y. Zhao. Sharing graphs using differentially private graph models. In *IMC*, 2011.

[14] Oliver Williams and Frank McSherry. Probabilistic inference and differential privacy. *Proc. NIPS*, 2010.