Edge sampling and graph parameter estimation via vertex neighborhood accesses

Tetek, Jakub; Thorup, Mikkel

Published in:
STOC 2022 - Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing

DOI:
10.1145/3519935.3520059

Publication date:
2022

Document version
Publisher's PDF, also known as Version of record

Document license:
CC BY

Citation for published version (APA):
Edge Sampling and Graph Parameter Estimation via Vertex Neighborhood Accesses

Jakub Tětek
Mikkel Thorup
jakub2thorup@gmail.com
University of Copenhagen
Copenhagen, Denmark

ABSTRACT

In this paper, we consider the problems from the area of sublinear-time algorithms of edge sampling, edge counting, and triangle counting. Part of our contribution is to consider three different settings, differing in the way in which one may access the neighborhood of a given vertex. In previous work, people have considered indexed neighbor access, with a query returning the \( i \)-th neighbor of a given vertex. Full neighborhood access model, which has a query that returns the entire neighborhood at a unit cost, has recently been considered in the applied community. Between these, we propose hash-ordered neighbor access, inspired by coordinated sampling, where we have a fully global random hash function, and can access neighbors in order of their hash values, paying a constant for each accessed neighbor.

For edge sampling and counting, our new lower bounds are in the most powerful full neighborhood access model. We provide matching upper bounds in the weaker hash-ordered neighbor access model. Our new faster algorithms can be provably implemented efficiently on massive graphs in external memory and with the current APIs for, e.g., Twitter or Wikipedia. For triangle counting, we provide a separation: a better upper bound with full neighborhood access than the known lower bounds with indexed neighbor access. The technical core of our paper is our edge-sampling algorithm on which the other results depend. We now describe our results on the classic problems of edge and triangle counting.

We give an algorithm that uses hash-ordered neighbor access to approximately count edges in time \( \tilde{O}(\frac{n}{\varepsilon \sqrt{m}} + \frac{1}{\varepsilon^2}) \) (compare to the state of the art without hash-ordered neighbor access of \( \tilde{O}(\frac{n}{\varepsilon^{2.5}}) \) by Eden, Ron, and Seshadhri [ICALP 2017]). We present an \( \Omega(\frac{n}{\varepsilon^{2.5}}) \) lower bound for \( \varepsilon \geq \sqrt{m}/n \) in the full neighborhood access model. This improves the lower bound of \( \Omega(\frac{n}{\varepsilon^{2.5}}) \) by Goldreich and Ron [Rand. Struct. Alg. 2008]) and it matches our new upper bound for \( \varepsilon \geq \sqrt{m}/n \). We also show an algorithm that uses the more standard assumption of pair queries ("are the vertices \( u \) and \( v \) adjacent?") with time complexity of \( \tilde{O}(\frac{n}{\varepsilon \sqrt{m}} + \frac{1}{\varepsilon^2}) \). This matches our lower bound for \( \varepsilon \geq \frac{m^{1/6}}{n^{1/3}} \).

Finally, we focus on triangle counting. For this, we use the full power of the full neighbor access. In the indexed neighbor model, an algorithm that makes \( \tilde{O}(\frac{n}{\varepsilon^{3.5}} + \min(m, \frac{m^{3/2}}{\varepsilon^2 T})) \) queries for \( T \) being the number of triangles, is known and this is known to be the best possible up to the dependency on \( \varepsilon \) (Eden, Levi, Ron, and Seshadhri [FOCS 2015]). We improve this significantly to \( \tilde{O}(\min(n, \frac{n}{\varepsilon^{2.5}} + \frac{\sqrt{m}}{\varepsilon^2 T})) \) full neighbor accesses, thus showing that the full neighbor access is fundamentally stronger for triangle counting than the weaker indexed neighbor model. We also give a lower bound, showing that this is the best possible with full neighborhood access, in terms of \( n, m, T \).

CCS CONCEPTS

• Theory of computation → Sketching and sampling.

KEYWORDS

Sublinear-time algorithms, Edge sampling, Edge counting, Triangle counting

ACM Reference Format:


1 INTRODUCTION

In this paper, we consider three well-studied problems from the area of sublinear-time algorithms: edge sampling, edge counting, and triangle counting in a graph. All of the three problems we consider have been studied before (in their approximate versions); see [10, 14] for edge sampling, [7, 12, 15, 17, 25] for edge counting, and [4, 8, 19, 22] for triangle counting. We first give an algorithm for exact edge sampling. We then apply this algorithm to both edge sampling and triangle counting in a graph. All of the three problems we consider have been studied before (in their approximate versions); see [10, 14] for edge sampling, [7, 12, 15, 17, 25] for edge counting, and [4, 8, 19, 22] for triangle counting. We first give an algorithm for exact edge sampling. We then apply this algorithm to both edge and triangle counting. We consider these three problems both in the well-studied indexed neighbor model, but also in two new models that we introduce.

The first of them is the full neighborhood access model. This model has recently been considered in the applied community

\[0]^{1}\text{In this setting, we have the following queries: given a vertex, return its degree; given a vertex }v\text{ and a number }i\leq d(v),\text{ return the }i\text{-th neighbor }v\text{ in an arbitrary ordering; return a random vertex.}\]
We give more efficient algorithms with the hash-ordered neighbor access. Specifically, we give the first sublinear-time algorithm that w.h.p. returns a sample of edges from exactly uniform distribution. It runs in expected time $O(\sqrt{s n} + 1)$, which is the same complexity as that from [9] for approximate edge sampling for constant $\epsilon = O(1)$ (we solve exact edge sampling, which is equivalent to the case $\epsilon = 0$). We give a near-matching lower bound for all choices of $n, m, s$. Apart from sampling with replacement, our methods also lead to algorithms for sampling without replacement and Bernoulli sampling.

Apart from being of interest in its own right, the problem of sampling multiple edges is also interesting in that it sheds light on the relationship between two standard models used in the area of sublinear algorithms. While many algorithms only use vertex accesses, many also use random edge sampling. An algorithm for uniform edge sampling then can be used to simulate random edge queries. Our algorithm is not only more efficient, but also has an advantage over the algorithm in [9] that it can be used as a black box. This is not possible with their algorithm as it only samples edges approximately uniformly.

We use this reduction between the settings with and without random edge queries in our new algorithm for triangle counting. We prove that, perhaps surprisingly, this reduction results in near-optimal complexity in terms of $n, m, T$ (where $T$ is the number of triangles). We also use our edge sampling algorithm for counting edges\(^4\), also resulting in a near-optimal complexity, this time even in terms of $\epsilon$, for $\epsilon \geq \sqrt{n}/m$.

Since we consider edge sampling to be the technical core of our paper, we focus on that in this extended abstract. We defer the rest of our results to the full version of the paper.

**Counting edges.** The problem of counting edges in sublinear time was first considered by Feige [15]. In his paper, he proves a new concentration inequality and uses it to give a $2 + \epsilon$ approximation algorithm for counting edges running in time $O\left(\frac{n}{\epsilon^3 \sqrt{m}}\right)$. This algorithm only uses random vertex and degree queries but no neighbor access. It is also proven in [15] that in this setting, $\Omega(n)$ time is required for $2 - \epsilon$ approximation for any $\epsilon > 0$.

Since we are dealing with a graph, it is natural to also consider a query that allows us to access the neighbors of a vertex. Goldreich and Ron [17] use indexed neighbor queries to break the barrier of 2-approximation and show a $(1 + \epsilon)$-approximation that runs in time $O\left(\frac{n}{\epsilon^3 \sqrt{m}}\right)$. They also present a lower bound of $\Omega\left(\frac{n}{\epsilon^3 \sqrt{m}}\right)$. To prove this lower bound, they take a graph with $m$ edges and add a clique containing $rm$ edges. To hit the clique with constant probability, $\Omega\left(\frac{n}{\epsilon^3 \sqrt{m}}\right)$ vertex samples are required.

Using a clever trick based on orienting edges towards higher degrees, Seshadhri [25] shows a much simpler algorithm. This approach has been later incorporated into the journal version of [8] and published in that paper. The trick of orienting edges also led to an algorithm for estimating moments of the degree distribution [12]. The moment estimation algorithm can estimate the number

\(^4\)Bernoulli sampling is defined as sampling each edge independently with some probability $p$.

\(^3\)We use $n$ and $m$ to denote the number of vertices and edges in the graph, respectively.

\(^2\)The authors claim complexity $O\left(\sqrt{s n} + s\right)$, which is sublinear in $s$. However, one clearly has to spend at least $O(s)$ time and perform $O(\min(m, s))$ queries.
of edges in time $\tilde{O}(\frac{n}{\epsilon \sqrt{m}})$ by estimating the first moment – the average degree. This is currently the fastest algorithm known for counting edges.

We show two more efficient algorithms that use either the pair queries or the hash-ordered neighbor access. Specifically, in this setting, we give an algorithm that approximately counts edges in time $\tilde{O}(\frac{n}{\epsilon \sqrt{m}} + \frac{1}{\epsilon^2})$. This bound is strictly better than the state of the art (assuming $\epsilon \ll 1$ and $m \ll n^2$). We also show that the (in some sense) slightly weaker setting\footnote{Any algorithm with pair queries with time/query complexity $Q$ can be simulated in $O(Q \log \log n)$ time/queries using hash-ordered neighbor access. We discuss this in Section 1.1} of indexed neighbor with pair queries ("are vertices $u$ and $v$ adjacent") is sufficient to get an algorithm with time complexity $\tilde{O}(\frac{n}{\epsilon \sqrt{m}} + \frac{1}{\epsilon^2})$. This improves upon the state of the art for $\epsilon$ being not too small. Our methods also lead to an algorithm in the indexed neighbor access setting that improves upon the state of the art for $\epsilon$ small enough.

We show lower bounds that are near-matching for a wide range of $\epsilon$. Specifically, we prove that $\Omega(\frac{n}{\epsilon \sqrt{m}})$ samples are needed for $\epsilon \geq \sqrt{m}/n$, improving in this range upon $\Omega(\frac{n}{\epsilon \sqrt{m}})$ from [17]. This lower bound holds not only with full neighbor access, but also in some more general settings. For example, Twitter API implements a query that also returns the degrees of the neighbors. Our lower bound also applies to that setting.

Triangle counting. The number of triangles $T$ in a graph can be trivially counted in time $O(n^3)$. This has been improved by Itai and Rodeh [19] to $O(m^{3/2})$. This is a significant improvement for sparse graphs. The first improvement for approximate triangle counting has been given by Kolountzakis et al. [22], who improved the time complexity to $\tilde{O}(m + \frac{m^{3/2}}{\epsilon^2 T})$ (recall that $T$ is the number of triangles). This has been later improved by Eden et al. [12] to $\tilde{O}(\frac{n}{\epsilon \sqrt{m} T} + \min(m, \frac{m^{3/2}}{\epsilon^2 T}))$. In that paper, the authors also prove that their algorithm is near-optimal in terms of $n, m, T$.

Variants of the full neighborhood access model are commonly used in practice, and the model has been recently used in the applied community [3]. Perhaps surprisingly, no algorithm performing asymptotically fewer queries than the algorithm by Eden et al. [12] is known in this setting. Since the number of neighborhood queries is often the bottleneck of computation (the rate at which one is allowed to make requests is often severely limited), more efficient algorithms in this model could significantly decrease the cost of counting triangles in many real-world networks. We fill this gap by showing an algorithm that performs $\tilde{O}(\min(n, \frac{n}{\epsilon \sqrt{m} T} + \frac{\min(m, \frac{m^{3/2}}{\epsilon^2 T})}{\epsilon^2 T}))$ queries. This is never worse than $\tilde{O}(\frac{n}{\epsilon \sqrt{m} T} + \frac{m^{3/2}}{\epsilon^2 T})$ and it is strictly better when $T \ll m^{9/4}/n^{3/2}$ or $\epsilon \ll 1$. This also improves the complexity in terms of $n, m, T$. Our result also proves a separation between the two models, as the algorithm by Eden et al. [12] is known to be near-optimal in terms of $n, m, T$ in the indexed neighbor model. Our triangle counting algorithm relies on our result for sampling edges, showcasing the utility of that result. Using the algorithm of Eden and Rosenbaum [14] to simulate the random edge queries would result in both worse dependency on $\epsilon$ and a more complicated analysis. We also prove near-matching lower bounds in terms of $n, m, T$.

**Setting without random vertex/edge queries.** If we are not storing the whole graph in memory, the problem of sampling vertices in itself is not easy. There has been work in the graph mining community that assumes a model where random vertex or edge queries are not available and we are only given a seed vertex. The complexity of the algorithms is then parameterized by an upper bound on the mixing time of the graph. The problem of sampling vertices in this setting has been considered by Ben-Eliezer et al. [3], Chierichetti and Haddadan [6]. The problems of approximating the average degree has been considered by Dasgupta et al. [7]. Counting triangles in this setting has been considered by Bera and Seshadhri [4].

### 1.1 What Is a Vertex Access?

**Motivation behind full neighborhood access.** All sublinear-time algorithms with asymptotic bounds on complexity published so far assume only a model which allows for indexed neighbor access. However, this model is usually too weak to model the most efficient ways of processing large graphs, as non-sequential access to the neighborhood is often not efficient or not possible at all. The full neighborhood access model attempts to capture this. For example, to access the $i$-th neighbor of a vertex in the Internet graph\footnote{Internet graph is a directed graph with vertices being webpages and a directed edge for each link.}, one has to, generally speaking, download the whole webpage corresponding to the vertex. Similarly, when accessing a real-world network through an API (this would usually be the case when accessing the Twitter graph, Wikipedia graph, etc.), while it is possible to get just the $i$-th neighbor of a vertex, one may often get the whole neighborhood at little additional cost. The reason is that the bottleneck is usually the limit on the allowed number of queries in some time period and in one query, one may usually get many neighbors. While there is usually a limit on this number of neighbors that can be fetched in one query, this limit is often large enough that for the vast majority of vertices, the whole neighborhood can be returned as a response to one query.

Although the above-mentioned APIs do not support random vertex queries, there are methods that implement the random vertex query and are efficient in practice [3, 5]. In the full neighborhood access model, we do not assume any specific vertex sampling method; any algorithm implementing the random vertex query may be used. This further strengthens the case for our model. If it takes several API calls to get one random vertex, one may perform multiple API calls on the neighborhood of each randomly sampled vertex without significantly increasing the complexity.

Another motivation for the full neighborhood access comes from graphs stored in external memory. When storing the graph on a hard drive, one may read $\approx 1-3$ MB in the same amount of time as the overhead caused by a non-linear access\footnote{Consider for example Seagate ST4000DM000 [24] and Toshiba MG07SCA14TA [26]. These are two common server hard drives. The average seek (non-linear access) times of these hard drives are 12 and 8.5 ms, respectively. Their average read speeds are 146 and 260 MB/s, respectively. This means that in the time it takes to do one seek, one can read $\approx 1.75$ and $\approx 2.2$ MB, respectively.}. If each neighbor is stored in roughly 10-30 bytes, then when we access one neighbor,
we may read on the order of \(10^5\) neighbors while increasing the running time only by a small factor of constant.

These considerations suggest that an algorithm in the full neighborhood access model with lower complexity may often be preferable to an asymptotically less efficient algorithm in the weaker indexed neighbor access setting. It is also for these reasons that this model has been recently used in the applied community [3].

**Lower bounds and the full neighbor access.** While some lower-bounds in the past have been (implicitly) shown in the full neighborhood access model (such as the one for edge counting in [17]), others do not apply to that setting (such as the one for triangle counting in [12]; in fact, we prove that it does not hold in the full neighborhood access model). As we have argued, there are many settings where one can easily get many neighbors of a vertex at a cost similar to getting one vertex. Lower bounds proven in the indexed neighbor access model then do not, in general, carry over to these settings. This highlights the importance of proving lower bounds in the full neighborhood model, which are applicable to such situations.

*Motivation behind hash-ordered neighbor access.* We introduce a model suitable for locally stored graphs, inspired by coordinated sampling. This model is also suitable for graphs stored in external memory. We call this model the hash-ordered neighbor access. It is an intermediate model, stronger than the indexed neighbor access but weaker than the full neighborhood access model.

While in the indexed neighbor model, the neighbors can be ordered arbitrarily, this is not the case with hash-ordered neighbor access. In this setting, we have a global hash function \(h: V \to (0, 1]\) which we may evaluate. Moreover, we assume that the neighbors of a vertex are ordered with respect to their hash values.

The simplest way to implement hash-ordered neighbor access is to store for each vertex its neighborhood in an array, sorted by the hash values. This has no memory overhead as compared to storing the values in an array. One may also efficiently support hash-ordered access on dynamic graphs using standard binary search trees.

We believe that the hash-value-ordered array is also a good way to implement pair queries when storage space is scarce. We may tell whether two vertices \(u\) and \(v\) are adjacent as follows. We evaluate \(h(u)\) and search the neighborhood of \(v\) for a vertex with this hash value. We use that the hash values are random, thus allowing us to use interpolation search. This way, we implement the pair query in time \(O(\log \log d(v)) \leq O(\log \log n)\) [1, 23].

We believe that the hash-ordered neighbor access can be useful for solving a variety of problems in sublinear time. Specifically, we show that it allows us to sample higher-degree vertices with higher probability—something that could be useful in other sublinear-time problems.

### 1.2 Our Techniques

In the part of this paper where we consider edge sampling, we replace each (undirected) edge by two directed edges in opposite directions. We then assume the algorithm is executed on this directed graph.

#### 1.2.1 Sampling One Edge by a Random Length Random Walk

The algorithm for sampling one edge is essentially the same as that for sampling an edge in bounded arboricity graphs from [11]. We use different parameters and a completely different analysis to get the logarithmic dependence on \(\varepsilon^{-1}\).

We call a vertex \(v\) heavy if \(d(v) \geq \theta\) and light otherwise, for some parameter \(\theta\) that is to be chosen later. A (directed) edge \(uv\) is heavy/light if \(u\) is heavy/light. Instead of showing how to sample edges from a distribution close to uniform, we show an algorithm that samples each (directed) edge with probability in \([1-\varepsilon)/2m, c/(2m)\) for some \(c > 0\) and fails otherwise (with probability \(\approx 1 - c\)). One may then sample an edge \(1 + \varepsilon\) pointwise close to random by doing in expectation \(1/c\) repetitions.

It is easy to sample light edges with probability exactly \(c'/(2m)\) — one may pick vertex \(v\) at random, choose \(j\) uniformly at random from \([\theta]\) and return the \(j\)-th outgoing edge incident to the picked vertex. Return “failure” if \(d(v) < j\). We now give an intuition on how we sample heavy edges.

We set \(\theta\) such that at least one half of the neighbors of any heavy vertex are light (we need a constant factor approximation of \(m\) for this; we use one of the standard algorithms to get it). Consider a heavy vertex \(v\). We use the procedure described above to sample a (directed) light edge \(uw\) and we consider the vertex \(w\). Since at least one half of the incoming edges of any heavy vertex are light, the probability of picking \(w\) is in \([c'd(o)/(4m), c'd(o)/(2m)]\). Sampling an incident edge, thus get that each heavy edge is sampled with probability in \([c'(4m), c'(2m)]\).

Combining these procedures for sampling light and heavy edges (we do not elaborate here on how to do this), we may sample edges such that for some \(c' > 0\) we sample each light edge with probability \(c'/2m\) while sampling each heavy edge with probability in \([c'(4m), c'(2m)]\) (the value \(c'\) is different from \(c\) due to combining the procedures for sampling light and heavy edges).

We now show how to reduce the factor of 2 to 1 + \(\varepsilon\). Consider a heavy vertex \(v\). Pick a directed edge \(uw\) from the distribution of the algorithm we just described and consider \(w\). The probability that \(w = v\) is in \([c'd(o)/(2m), c'd'(o)/(2m)]\), as we now explain. Let \(h_o\) be the fraction of neighbors of \(v\) that are heavy. Light edges are picked with probability \(c'(2m)\) and at least half of the incoming edges are light. The remaining edges are picked with probability in \([c'(4m), c'(2m)]\). The probability of sampling \(v\) is then \(\geq (1 - h_o)c'(2m) + h_oc'/4m \geq 3c'_o/8m\) because \(h_o \leq 1/2\). The probability is also clearly \(\leq c'(2m)\), thus proving the claim. Combining edge sampling and heavy edge sampling (again, we do not elaborate here on how to do this), we are now able to sample an edge such that each light edge is sampled with probability \(c''/(2m)\) and each heavy edge with probability in \([3c''/8m, c''/2m]\) (again, the value \(c''\) is different from \(c, c'\) due to combining light and heavy edge sampling). Iterating this, the distribution converges pointwise to uniform at an exponential rate.

One can show that this leads to the following algorithm based on constrained random walks of random length. The length is chosen uniformly at random from \([k]\) for integer \(k \approx \log c^{-1}\). The algorithm returns the last edge of the walk. The random walk has constraints that, when not satisfied, cause the algorithm to fail and restart. These constraints are (1) the first vertex \(v\) of the walk is light and
Table 1: Comparison of our results with previous work. Note that the empty cells in the table do not imply that nothing is known about the problems — an algorithm that works in some model can also be used in any stronger model and similarly a lower bound that holds in some model also holds in any weaker model. We show the previous lower bounds as holding for our newly defined models when they straightforwardly extend to the setting. Any problem can be trivially solved in $O(n + m)$. We do not make this explicit in the bounds. Similarly, any stated lower bound is assumed to hold in the sublinear regime, unless specified otherwise.

<table>
<thead>
<tr>
<th></th>
<th>Upper bound previous work</th>
<th>Upper bound this paper</th>
<th>Lower bound previous work</th>
<th>Lower bound this paper</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sampling one edge</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indexed neighbor</td>
<td>$O\left( \frac{m}{\sqrt{m}} \right)$</td>
<td>$O\left( \frac{m\log^c n}{\sqrt{m}} \right)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
<td></td>
</tr>
<tr>
<td>Indexed neighbor</td>
<td>$O\left( \sqrt{n} \frac{1}{\sqrt{m}} + s \right)$</td>
<td>$O\left( \sqrt{n} \frac{1}{\sqrt{m}} \right)$ for $m \geq \Omega(n)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} + s \right)$</td>
<td>$\Omega\left( \sqrt{n} \frac{1}{\sqrt{m}} + s \right)$</td>
</tr>
<tr>
<td><strong>Sampling 5 edges</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indexed neighbor access</td>
<td>$O\left( \sqrt{nm} \frac{1}{\sqrt{m}} + \frac{s}{\sqrt{m}} \right)$</td>
<td>$O\left( \sqrt{nm} \frac{1}{\sqrt{m}} \right)$ for $m \geq \Omega(n)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} + s \right)$</td>
<td>$\Omega\left( \sqrt{n} \frac{1}{\sqrt{m}} + s \right)$</td>
</tr>
<tr>
<td><strong>Edge counting</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indexed neighbor</td>
<td>$O\left( \frac{m}{\sqrt{m}} \right)$</td>
<td>$O\left( \frac{m}{\sqrt{m}} + \frac{s}{\sqrt{m}} \right)$ for $m \geq \Omega(n)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
</tr>
<tr>
<td>Indexed neighbor</td>
<td>$O\left( \frac{m}{\sqrt{m}} \right)$</td>
<td>$O\left( \frac{m}{\sqrt{m}} + \frac{s}{\sqrt{m}} \right)$ for $m \geq \Omega(n)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
</tr>
<tr>
<td>Full neighborhood access w/ random vertex query</td>
<td>$O\left( \frac{m}{\sqrt{m}} + \frac{\log n}{\sqrt{m}} \right)$</td>
<td>$O\left( \frac{m}{\sqrt{m}} + \frac{\log n}{\sqrt{m}} \right)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
</tr>
<tr>
<td>Full neighborhood access w/ random edge query</td>
<td>$O\left( \frac{m}{\sqrt{m}} + \frac{\log n}{\sqrt{m}} \right)$</td>
<td>$O\left( \frac{m}{\sqrt{m}} + \frac{\log n}{\sqrt{m}} \right)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
</tr>
<tr>
<td>Full neighborhood access w/ random vertex query</td>
<td>$O\left( \frac{m}{\sqrt{m}} + \frac{\log n}{\sqrt{m}} \right)$</td>
<td>$O\left( \frac{m}{\sqrt{m}} + \frac{\log n}{\sqrt{m}} \right)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
</tr>
<tr>
<td>Full neighborhood access w/ random edge query</td>
<td>$O\left( \frac{m}{\sqrt{m}} + \frac{\log n}{\sqrt{m}} \right)$</td>
<td>$O\left( \frac{m}{\sqrt{m}} + \frac{\log n}{\sqrt{m}} \right)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
<td>$\Omega\left( \frac{m}{\sqrt{m}} \right)$</td>
</tr>
</tbody>
</table>

all subsequent vertices are heavy except the last one (which may be either light or heavy) and (2) picking $X \sim \text{Bern}(d(v) / \sqrt{2m})$, the first step of the walk fails if $X = 0$ (note that this is equivalent to using rejection sampling to sample the first edge of the walk).

1.2.2 **Sampling Multiple Edges and Edge counting using Hash-Ordered Neighbor Access**. The problems of edge counting and sampling share the property that in solving both these problems, it would be of benefit to be able to sample vertices in a way that is biased towards vertices with higher degree. This is clearly the case for edge sampling as it can be easily seen to be equivalent to sampling vertices with probabilities proportional to their degrees. Biased sampling is also useful for edge counting. If we let $v$ be chosen at random from some distribution and for a fixed vertex $u$ define $P_v = P(v = u)$, then $X = \frac{d(v)}{P_v}$ is an unbiased estimate of the number of edges, called the Horvitz-Thompson estimator [18]. The variance is $\text{Var}(X) \leq E(X^2) = \sum_{v \in V} P_v \left( \frac{d(v)}{P_v} \right)^2$ which decreases when high-degree vertices have larger probability $P_v$. The crux of this part of our paper, therefore, lies in how to perform this biased sampling.

**Biased sampling procedure.** We first describe how to perform biased sampling in the case of the indexed neighbor access model and then describe a more efficient implementation in the hash-ordered neighbor access model. We say a vertex $v$ is heavy if $d(v) \geq \theta$ for some value of $\theta$ and we say it is light otherwise. The goal is to find all heavy vertices in the graph as that will allow us to sample heavy vertices with higher probability. In each step, we sample a vertex and look at its neighborhood. This takes in expectation $O(m/n)$ time per step. After $\Theta(n \log(n)/\theta)$ steps, we find w.h.p. all vertices with degree at least $\theta$. We call this technique high-degree exploration.

How to exploit hash-ordered neighbor access? We do not actually need to find all heavy vertices in order to be able to sample from them when we have the vertex hash queries. We make a sample $S$ of vertices large enough such that, w.h.p., each heavy vertex has one of its neighbors in $S$. Pick all heavy vertices incident to the sampled vertices that have $h(v) \leq \theta$. We can find these vertices in constant time per vertex using hash-ordered access. Since each heavy vertex has at least one of its neighbors in $S$, these are in fact all heavy vertices $v$ with $h(v) \leq \theta$. Since the hash values are independent and uniform on $[0, 1]$, each heavy vertex is (w.h.p.) picked independently with probability $\theta$. This allows us to sample heavy vertices with larger probability than if sampling uniformly.

We apply this trick repeatedly for $k = 1, \ldots, \log n$ with thresholds $\theta_k = 2^k \theta$ and $P_k = 2^k \theta$. This way, instead of having one threshold $\theta$ and one probability of being sampled for each $v$ with
We find it using a geometric search.

**Edge sampling.** We describe how to sample each edge independently with some probability $p$. This setting is called Bernoulli sampling. Light edges can be sampled in a way similar to that used for sampling light edges in the algorithm for sampling one edge. Using the biased sampling algorithm allows us to sample heavy vertices with higher probability. We then prove that for each vertex $v$, the probability that $v$ is sampled is at least the probability that one of the edges incident to $v$ is sampled when sampling each edge independently with probability $p$. This allows us to use rejection sampling to sample each vertex $v$ with a probability equal to the probability of at least one of its incident edges being sampled. By sampling $k$ incident edges of each vertex where $k$ is chosen from the right distribution, we get that each heavy edge is sampled independently with some fixed probability $p$. Sampling light and heavy edges separately and taking union of those samples gives us an algorithm that performs Bernoulli sampling from the set of all edges.

One can use Bernoulli sampling to sample edges without replacement. Specifically, when it is desired to sample $s$ edges without replacement, one may use Bernoulli sampling to sample in expectation, say, $2s$ samples and if at least $s$ are sampled, then return a random subset of the random edges, otherwise repeat.

To sample $s$ edges with replacement, we perform Bernoulli sampling $\Theta(s)$ times, setting the probability such that each time, we sample in expectation $O(1)$ edges. From each (non-empty) Bernoulli sample, we take one edge at random and add it into the sample. While implementing this naively would result in a linear dependence on $s$, this can be prevented. The reason is that it is sufficient to perform the pre-processing for Bernoulli sampling only once. This allows us to spend more time in the "high-degree exploration phase", making the Bernoulli sampling itself more efficient.

**Edge counting by sampling.** When using Bernoulli sampling with some inclusion probability $p$, the number of sampled edges $|S|$ is concentrated around $pm$. We estimate $m$ as $|S|/p$ and prove that for an appropriate choice of $p$, the approximation has error at most $\epsilon$ with high probability. The "appropriate value" of $p$ depends on $m$. We find it using a geometric search.

**Counting edges directly.** We also give an independent algorithm for edge counting based on a different idea. We use the biased sampling procedure to sample higher-degree vertices with higher probability. We then use the above-mentioned Horvitz-Thompson estimator; sampling higher-degree vertices with higher probability reduces the variance. We then take the average of an appropriate number of such estimators to sufficiently further reduce the variance.

**1.2.3 Edge Counting Using Pair Queries.** Seshadhri [25] shows a bound on the variance of the following estimator: sample a vertex $v$, get a random neighbour $u$ of $v$, if $(d(v), id(v)) < (d(u), id(u))$ then answer $nd(v)$, otherwise answer 0. This is an unbiased estimate of $m$. We combine this idea with the technique of high-degree exploration. Direct all edges from the endpoint with lower degree to the one with higher degree. The biggest contributor to the variance of the estimator from [25] are the vertices that have out-degree roughly $\sqrt{m}$ (one can show that there are no higher-out-degree vertices). Our goal is to be able to sample these high out-degree $(d^+(v) \geq \theta)$ for some parameter $\theta$ vertices with higher probability. Using the Horvitz-Thompson estimator for vertices of out-degree at least $\theta$ will decrease the variance. We use an estimator inspired by the one from [25] for vertices with out-degree $< \theta$.

We make a sample $S$ of vertices and let $S'$ be the subset of $S$ of vertices having degree $\geq \theta$. The intuition for why we consider $S'$ is the following. If we pick $S$ to be large enough ($|S| = \Theta(n \log(n)/\theta)$ is sufficient), any vertex $v$ with $d^+(v) \geq \theta$ will have at least one of its out-neighbors sampled in $S$. Moreover, it holds $d(v) \geq d^+(v) \geq \theta$; since we direct edges towards higher-degree endpoints, these sampled out-neighbors also have degree $\geq \theta$. This means that they also lie in $S'$. Any high-out-degree vertex thus has, w.h.p., a neighbor in $S'$. At the same time, $S'$ has the advantage of being significantly smaller than $S$ as there can only be few vertices with degree $\geq \theta$ in the graph (at most $2m/\theta$, to be specific), so each vertex of $S$ lies in $S'$ with probability $\leq 2m/(n\theta)$.

Now we pick each incident edge to $S'$ with a fixed probability $p$ and for each picked edge $uv$ for $u \in S'$, mark the vertex $v$. A vertex $v$ is then marked with probability $p_0 = 1 - (1 - p)^{d^+(v)}$ where $r(v) = |N(v) \cap S'|$. It then holds $p_0 \approx p r(v)$ (we ensure $r(v)$ is not too large, which is needed for this to hold). This is (w.h.p.) roughly proportional to the out-degree of $v$. Using the Horvitz-Thompson estimator, this reduces the variance. This suffices to get the improved complexity.

It remains to show how to efficiently compute $r(v)$ (we need to know the value in the Horvitz-Thompson estimator). We set the threshold $\theta$ so as to make sure that only a small fraction of all vertices can have degree greater than the threshold. Then, $S'$ will be much smaller than $S$ (as $S'$ is a uniform sample). In fact, it will be so small that we can afford to use pair queries to check which of the vertices in $S'$ are adjacent to $v$. This is the main trick of our algorithm.

Another obstacle that we have to overcome is that when we are given a vertex, we cannot easily determine its in-degree. We need to know this to decide which of the above estimators to use for the vertex. Fortunately, the cost of estimating it is roughly inverse to the probability we will need the estimate, thus making the expected cost small.

**1.2.4 Triangle Counting Using Full Neighborhood Access.**

Warmup: algorithm with random edge queries. We now show a warmup which assumes that we may sample edges uniformly at random. We then use this as a starting point for our algorithm. This warmup is inspired by and uses some of the techniques used in [20].

Consider an edge $e = uv$. By querying both endpoints, we can determine the number of triangles containing $e$ (because this number is equal to $|N(u) \cap N(v)|$). Let $t(e)$ be the number of triangles containing $e$. We are assuming $id$ is a bijection between $V$ and $[n]$ and $<$ on the tuples is meant with respect to the lexicographic ordering.

\[^{10}\text{We are assuming } id \text{ is a bijection between } V \text{ and } [n] \text{ and } < \text{ on the tuples is meant with respect to the lexicographic ordering.}\]
containing \( e \). One of the basic ideas that we use is that we assign each triangle to its edge with the smallest value \( t(e) \). This trick has been used before for edge counting in [25] and for triangle counting in [20]. We denote by \( t^*(e) \) the number of edges assigned to \( e \). Consider a uniformly random edge \( e = uv \) and a uniform vertex \( w \in N(u) \cap N(v) \). Let \( X = t(e) \) if \( uvw \) is assigned to \( uv \) and \( X = 0 \) otherwise. The expectation is \( E(X) = \sum_{e \in E} \frac{1}{m} \cdot \frac{t^*(e)}{t(e)} \cdot t(e) = T/m \) and we may thus give an unbiased estimator of \( T \) as \( mX \). The variance of \( X \) is \( Var(X) \leq \sum_{e \in E} \frac{1}{m} \cdot \frac{t^*(e)}{t(e)} \cdot (t(e))^2 = \frac{1}{m} \sum_{e \in E} t^*(e) \cdot t(e) \). A bound used in Kallaugher et al. [20] can be proved to find this is \( O(T^{4/3}/m) \). Taking \( s = \Theta(m/(\epsilon^2 T^{2/3})) \) samples and taking the average then gives a good estimate with probability at least \( 2/3 \) by the Chebyshev inequality.

We prove that this is optimal up to a constant factor in terms of \( m \) and \( T \) when only random edge queries (and not random vertex queries) are allowed. We will now consider this problem in the full neighborhood access model, which only allows for random vertex queries (and not random edge queries). Combining this algorithm with our edge sampling algorithm results in complexity \( O(\sqrt{n/m} + \frac{n}{\epsilon^2 T^{1/3}}) \) in the full neighborhood access model. This, however, is not optimal.

**Sketch of our algorithm: algorithm with random vertex queries.** We will now describe a more efficient algorithm that uses both random edge and random vertex queries. We then remove the need for random edge queries by simulating them with our algorithm for edge sampling. Perhaps surprisingly, black-box application of our edge sampling algorithm results in near-optimal complexity.

In order to find out the value \( t(e) \) for as many edges as possible, one may sample each vertex independently with some probability \( p \). For any edge \( e \) whose both endpoints have been sampled, we can compute \( t(e) \) with no additional queries. The sum of \( t(e) \)'s that we learn in this way is in expectation \( 3p^2T \). We may thus get an unbiased estimator of \( T \). We may express the variance by the law of total variance and a standard identity for the variance of a sum. By doing this, we find out that there are two reasons the variance is high. First, the variance of the number of triangles contributed to the estimate by one edge can be large\(^{11}\). This is true for edges that are contained in relatively many triangles. The second reason is the correlations between edges: if we have edges \( e, e' \) sharing one vertex and both endpoints of \( e \) have been sampled, it is more likely that both endpoints of \( e' \) are sampled, too. This introduces correlation between the contributions coming from different edges, thus increasing the variance. We now sketch a solution to both these issues.

The first issue could be solved by applying the above-described trick with assigning each triangle to its edge \( e = uv \) with the smallest value \( t(e) \). Pick \( w \) uniformly from \( N(u) \cap N(v) \) and let \( X_e = t(e) \) if \( uvw \) is assigned to \( uv \) and let \( X_e = 0 \) otherwise. An analysis like the one described above would give good bounds on the variance of \( X_e \). The issue with this is that for each edge with non-zero \( t(e) \), we query the vertex \( w \). If the number of edges in the subgraph induced by the sampled edges is much greater than the number of sampled vertices, this will significantly increase the query complexity. To solve this issue, we separately consider two situations. If an edge has many triangles assigned to it, we do the following. We sample a set of vertices \( N \) (this set is shared for all edges) large enough such that, by the Chernoff bound, \( \{\text{we} \in N(u) \cap N(v), \text{s.t.} uvw \text{ is assigned to } uv\} \approx t^*(e)/t(e) \). Instead of sampling \( w \) uniformly from \( N(u) \cap N(v) \), we then sample from \( N \cap N(u) \cap N(v) \). Since we do not need to query any vertex twice, we may bound the cost of this by \( |N| \). On the other hand, if the number of triangles assigned to \( e \) is small, the variance of \( X_e \) (which is proportional to \( t^*(e)/t(e) \)) is relatively small (as \( t^*(e) \) is small). We then may afford to only use the estimator \( X_e \) with some probability \( p' \) and if we do, we use \( X_e/p' \) as the (unbiased) estimate of \( t^*(e) \). This increases the variance contributed by the edge \( e \), but we may afford this as it was small before applying this trick. This way, we have to only make an additional query with probability \( p' \), thus decreasing the query complexity of this part of the algorithm.

The second problem is created by vertices whose incident edges have many triangles (at least \( \theta \)) assigned to them (as becomes apparent in the analysis). A potential solution would be to not use this algorithm for the vertices with more than \( \theta \) triangles assigned to incident edges and instead estimate the number of these triangles by the edge-sampling-based algorithm from the warm-up. Why is this better than just using that algorithm on its own? There cannot be many such problematic vertices. Namely, there can only be \( \epsilon T/\theta \) such vertices. This allows us to get a better bound on the variance. Specifically, instead of bounding the variance by \( O(T^{4/3}/m) \), we prove a bound of \( O(T/m) \). An issue we then have to overcome is that we cannot easily tell apart the "heavy" and "light" vertices. Let us have a vertex \( v \) and we want to know whether it is light (it has at most \( \theta \) triangles assigned to its incident edges) or whether it is heavy. The basic idea is to sample some vertices (this set is common for all vertices) and then try to infer whether a vertex is with high probability light or whether it may potentially be heavy based on the number of triangles containing edges between \( v \) and this set of sampled vertices.

These techniques lead to a bound of \( \tilde{O}(\frac{n}{\epsilon^2 T^{1/3}} + \sqrt{\frac{nm}{\epsilon^2 T}}) \). One can always read the whole graph in \( n \) full neighborhood queries, leading to a bound of \( O(\min(n, \frac{n}{\epsilon^2 T^{1/3}} + \frac{nm}{\epsilon^2 T})) \).

**Lower bounds.** Our lower bounds match our algorithms in terms of dependency on \( n, m, T \). The lower bound of \( \Omega(n/T^{1/3}) \) is standard and follows from the difficulty of hitting a clique of size \( T^{1/3} \). We thus need to prove a lower bound of \( \Omega(\min(n, \sqrt{nm/T})) \). This amounts to proving \( \Omega(\sqrt{nm/T}) \) under the assumption \( T \geq n \). We thus assume for now this inequality.

Our lower bound is by reduction from the OR problem (given booleans \( x_1, \ldots, x_n \), compute \( \bigvee_{i=1}^n x_i \)) in a style similar to the reductions in [13]. The complexity of the OR problem is \( \Omega(n) \). For an instance of the OR problem of size \( \sqrt{nm/T} \), we define a graph \( G \) with \( \Theta(n) \) vertices and \( \Theta(m) \) edges. The number of triangles is either \( \geq T \) if \( \bigvee_{i=1}^n x_i = 1 \) or \( 0 \) if \( \bigvee_{i=1}^n x_i = 0 \). Moreover, any query on \( G \) can be answered by querying one \( x_i \) for some \( i \in [n] \). It follows that any algorithm that solves triangle counting in \( G \) in \( Q \) queries can be used to solve the OR problem of size \( \Theta(\sqrt{nm/T}) \) in \( Q \) queries. This proves the desired lower bound.

\(^{11}\)Formally, we are talking about the variance of a random variable \( X_e \) equal to \( t(e) \) if both endpoints of \( e \) are sampled and \( 0 \) otherwise.
We now describe the graph $G$. We define a few terms. A section consists of $4$ groups of $\sqrt{nT/m}$ vertices. The whole graph consists of sections and $m/n$ non-section vertices. There are $\sqrt{nm/T}$ sections, one for each $x_i$. In the $i$-th section, there is a complete bipartite graph between the first two groups of vertices if $x_i = 0$ and between the third and fourth if $x_i = 1$. There is a complete bipartite graph between each third or fourth group of a section and the non-section vertices. If $x_i = 0$ for all $i$, then $G$ is triangle-free. If $x_i = 1$, then the $i$-th section together with the non-section vertices forms $\sqrt{nT/m} \cdot \sqrt{nT/m} \cdot m/n = T$ triangles. At the same time, a query “within a section” only depends on the value $x_j$ corresponding to that section, so we can implement it by one query to the instance of the OR problem. A query that does not have both endpoints within the same section is independent of the OR problem instance. The number of vertices and edges is $\Theta(1123)$ vertices. If $x_i$’s of all neighbors of the vertices are assumed to be ordered adversarially. Moreover, the algorithm is by the following queries:

- For $i \in [n]$, return the $i$-th vertex in the graph
- For $v \in V$, return $d(v)$
- For $v \in V$ and $i \in [d(v)]$, return the $i$-th neighbor of $v$
- For a given vertex $v$, return $id(v)$ such that if $v$ is the $i$-th vertex, then $id(v) = i$

where the vertices in the graph as well as the neighbors of a vertex are assumed to be ordered adversarially. Moreover, the algorithm is assumed to know $n$. This definition is standard; see [17] for more details. Pair queries are often assumed to be available in addition to the queries described above:

- Given vertices $u, v$, return whether the two vertices are adjacent
- For $v \in V$, return $h(v)$

This has been used, for example, in [2, 12, 14].

In this paper, we introduce a natural extension of the above-described setting without pair queries, which we call the hash-ordered neighbor access model. In this model, there is the following additional query

- For $v \in V$, return $h(v)$

where the hash of $v$, denoted $h(v)$, is a number picked independently uniformly at random from $[0, 1]$. Moreover, neighborhoods of vertices are assumed to be ordered with respect to the hashes of the vertices. Our algorithms do not require the vertices to be ordered with respect to the hash values (in contrast to the neighborhoods), although that would also be a natural version of this model.

We define the full neighborhood access model as follows. Each vertex $v$ has a unique $id(v) \in [n]$. We then have one query: return the $id$’s of all neighbors of the $i$-th vertex. We then measure the complexity of an algorithm by the number of queries performed, instead of the time complexity of the algorithm.

### 2.2 Pointwise $\varepsilon$-Approximate Sampling

**Definition 1.** A discrete probability distribution $P$ is said to be pointwise $\varepsilon$-close to $Q$ where $P, Q$ are assumed to have the same support, denoted $|P - Q|_p \leq \varepsilon$, if

$$|P(x) - Q(x)| \leq \varepsilon Q(x), \text{ or equivalently } 1 - \varepsilon \leq \frac{P(x)}{Q(x)} \leq 1 + \varepsilon$$

for all $x$ from the support.

In this paper, we consider distributions pointwise $\varepsilon$-close to uniform. This measure of similarity of distributions is related to the total variational distance. Specifically, for any $P, Q$, it holds that $|P - Q|_{TV} \leq |P - Q|_p$ [14].

### 2.3 Conditioning Principle

Let $X, Y$ be two independent random variables taking values in a set $A$ and let $f$ be a function on $A$. If $Y \sim f(X)$, then $X \sim f(Y)$. In other words, if we want to generate a random variable $X$ from some distribution, it is sufficient to be able to generate (1) a random variable from the distribution conditional on some function of $X$ and (2) a random variable distributed as the function of $X$. We call this the conditioning principle. We often use this to generate a sample - we first choose the sample size from the appropriate distribution and then sample the number of elements accordingly.

### 2.4 Notation

We use relations $f(x) \leq g(x)$ with the meaning that $f(x)$ is smaller than $g(x)$ up to a constant factor. The relations $\gg, \sim$ are defined analogously. The notation $f(x) \sim g(x)$ has the usual meaning of $f(x)/g(x) \to 1$ for $x \to \infty$. We use $\log x$ to denote the binary logarithm of $x$. We use $N(o)$ to denote the set of neighbors of $v$. Given a vertex $v$ and integer $i$, we let $i[v]$ to be the $i$-th neighborhood of $v$. Given a (multi-)set of vertices $S$, we let $d(S) = \sum_{v \in S} d(v)$ and $d_S(v) = |N(v) \cap S|$. For an edge $e = uv$, we denote by $N(e)$ the set of edges incident to either $u$ or $v$. In addition to sampling with and without replacement, we use the less standard name of Bernoulli sampling. In this case, each element is included in the sample independently with some given probability $p$ which is the same for all elements. Given a priority queue $Q$, the operation $Q.top()$ returns the elements with the lowest priority. $Q.pop()$ returns the element with the lowest priority and removes it from the queue.

We use $\text{Bern}(p)$ to denote a Bernoulli trial with bias $p$. $\text{Unif}(a, b)$ to be the uniform distribution on the interval $[a, b]$, $\text{Bin}(n, p)$ to be the binomial distribution with universe size $n$ and sample probability $p$. For distribution $D$ and event $E$, we use $X \sim (D|E)$ to denote that $X$ is distribution according to the conditional distribution $D$ given $E$.

### 3 Edge Sampling

We start with some definitions which we will be using throughout this section. Given a threshold $\theta$ (the exact value is different in each algorithm), we say a vertex $v$ is heavy if $d(v) \geq \theta$ and light if $d(v) < \theta$. We denote the set of heavy (light) vertices by $V_H$ ($V_L$). In this section, we replace each unoriented edge in the graph by two oriented edges in opposite directions. We then assume the algorithm is executed on this oriented graph. We then call a (directed) edge $uv$ heavy (light) if $u$ is heavy (light). If we can sample edges from this...
oriented graph, we can also sample edges from the original graph by sampling an edge and forgetting its orientation.

3.1 Sampling One Edge in the Indexed Neighbor Access Model

In this section, we show Algorithm 2 which samples an edge pointwise $\epsilon$-approximately in expected time $O(\frac{\sqrt{m}}{n \log^2 n})$. This algorithm is, up to a change of parameters, the one used in [11] but we provide a different analysis that is tighter in the case of general graphs (in [11], the authors focus on graphs with bounded arboricity). This algorithm works by repeated sampling attempts, each succeeding with probability $\approx \frac{\sqrt{m}}{n \log^2 n}$. We then show that upon successfully sampling an edge, the distribution is pointwise $\epsilon$-close to uniform.

**Algorithm 1:** Sampling_attempt($k$) subroutine

1. $\theta \leftarrow \lceil \sqrt{2m} \rceil$
2. Sample a vertex $u_0 \in V$ uniformly at random
3. If $u_0$ is heavy, return "failure"
4. Choose a number $j \in [\theta]$ uniformly at random
5. Let $u_1$ be the $j$'th neighbor of $u_0$; return "failure" if $d(u_0) < j$
6. For $i$ from 2 to $k$
   7. If $u_{i-1}$ is light, return "failure"
   8. $u_i \leftarrow$ random neighbor of $u_{i-1}$
7. Return $(u_{k-1}, u_k)$.

**Algorithm 2:** Sample an edge from distribution pointwise $\epsilon$-close to uniform

1. Pick $k$ from $\{1, \ldots, \ell = \lceil \sqrt{\frac{m}{2}} \rceil + 2\}$ uniformly at random
2. Call Sampling_attempt($k$), if it fails, go back to line 1, otherwise return the result

We show separately for light and heavy edges that they are sampled almost uniformly. The case of light edges (Observation 2) is analogous to the proof in [14]. We include it here for completeness.

**Observation 2.** For $k$ chosen uniformly from $[\ell]$, any fixed light edge $e = uv$ is chosen by Algorithm 1 with probability $\frac{1}{mr}$.

Proof. The edge $uv$ is chosen exactly when $k = 1$ (this happens with probability $\frac{1}{2}$), $u_0 = u$ (happens with probability $\frac{1}{2}$), and $j$ is such that $v$ is the $j$-th neighbor of $u$ (happens with probability $\frac{1}{2}$). This gives total probability of $\frac{1}{2mr}$.

We now analyze the case of heavy edges. Before that, we define for $v$ being a heavy vertex $h_{0,i} = \frac{d_H(v)}{d_H(v)}$ and for $i \geq 2$

$$h_{0,i} = h_{0,1} \sum_{w \in N_H(v)} h_{w,i-1} d_H(v)$$

For light vertices, the $h$-values are not defined.

**Lemma 3.** For $k$ chosen uniformly from $[\ell]$, for any heavy edge $uv$

$$P(u_{k-1} = v, u_k = w | k \geq 2) = (1 - h_{0,\ell-1}) \frac{1}{(\ell - 1)n\theta}$$

Proof. Let $k$ be chosen uniformly at random from $\{2, \ldots, r\}$. We show by induction on $r$ that $P(u_{k-1} = v | r) = (1 - h_{0,r-1}) \frac{d(v)}{(r-1)n\theta}$ for any heavy vertex $v$. If we show this, the lemma follows by substituting $r = \ell$ and by uniformity of $u_k$ on the neighborhood of $u_{k-1}$.

For $r = 2$, the claim holds because when $k = 2$, there is probability $\frac{1}{n\theta}$ that we come to $v$ from any of the $(1 - h_{0,1})d(v)$ adjacent light vertices.

We now show the induction step. In the following calculation, we denote by $P_r(E)$ the probability of event $E$ when $k$ is chosen uniformly from $[r]$. Consider some vertex $v$ and take a vertex $w \in N(v)$. It now holds $P_r(u_{k-2} = w) = P_{r-1}(u_{k-2} = w)$. We have

$$P_r(u_{k-1} = v) = \sum_{w \in N(v)} P_r(u_{k-2} = w)P(u_{k-1} = v | u_{k-2} = w)$$

$$= \sum_{w \in N(v)} P(u_0 = w)P(u_1 = v | u_0 = w)P(k = 2)$$

$$+ \sum_{w \in N_H(v)} P_{r-1}(u_{k-1} = w | k > 1)P(u_k = v | u_{k-1} = w)P(k > 2)$$

$$= \sum_{w \in N(v)} \frac{1}{n\theta} \frac{1}{r-1} + \sum_{w \in N_H(v)} (1 - h_{w,r-2}) \frac{d(v)}{(r-2)n\theta} \frac{1}{d(w)} \frac{r-2}{r-1}$$

$$= (1 - h_{0,2}) \frac{d(v)}{(r-1)n\theta} + \sum_{w \in N_H(v)} (1 - h_{w,r-2}) \frac{1}{(r-1)n\theta}$$

$$= (1 - h_{0,1}) \frac{d(v)}{(r-1)n\theta} + h_{0,1}d(v) \frac{1 - \sum_{w \in N_H(v)} h_{w,r-2} / d_H(v)}{(r-1)n\theta}$$

$$= (1 - h_{0,1}) \frac{d(v)}{(r-1)n\theta} + (h_{0,1} - h_{0,r-1}) \frac{d(v)}{(r-1)n\theta}$$

$$= \frac{d(v)}{(r-1)n\theta} \frac{1}{d_H(v)}$$

By the induction hypothesis, the lemma holds when $r = 2$.

Before putting it all together, we will need the following bound on $h_{0,1}$.

**Lemma 4.** For any $v \in V_H(G)$ and $k \geq 1$ it holds that

$$h_{0,k} \leq 2^{-k}$$

Proof. We first prove that for any $v \in V(G)$, it holds that $h_{0,1} \leq 1/2$. This has been shown in [14] and we include this for completeness. We then argue by induction that this implies the lemma.

Since $v$ is heavy, it has more than $\theta$ neighbors. Moreover, there can be at most $m$ heavy vertices, meaning that the fraction of heavy neighbors of $v$ can be bounded as follows

$$h_{0,1} \leq \frac{m}{\theta} \frac{1}{\sqrt{2m}} \leq \frac{1}{2}$$
We now show the claim by induction. We have shown the base case and it therefore remains to prove the induction step:

\[ h_{\alpha,\ell} = h_{\alpha,1} \sum_{w \in \mathcal{E}_H(v)} h_{\alpha,\ell-1} / d_H(v) \leq \frac{1}{2} \sum_{w \in \mathcal{E}_H(v)} 2^{-(i-1)} / d_H(v) = 2^{-i} \]

We can now prove the following theorem

**Theorem 5.** For \( \epsilon \leq \frac{1}{2} \), the Algorithm 2 runs in expected time \( O\left(\frac{n}{\sqrt{m}} \log \frac{1}{\epsilon}\right) \) and samples an edge from a distribution that is pointwise \( \epsilon \)-close to uniform.

**Proof.** We first prove the correctness and then focus on the time complexity.

**Correctness.** We first show that in an iteration of Algorithm 2, each edge is sampled with probability in \([1 - (1 - \epsilon) \frac{1}{m} ] \). For light edges, this is true by Observation 2. We prove the same for heavy edges. Similarly, a heavy edge \((v, w)\) is chosen when \( k \geq 2 \), \( u_{k-1} = v \) and \( w = u_k \). Using Lemma 3,

\[ P(k \geq 2, u_{k-1} = v, u_k = w) = P(k \geq 2)P(u_{k-1} = v, u_k = w | k \geq 2) \]

\[ \geq (1 - 2\epsilon + \epsilon^2) \frac{1}{\lfloor \ell \rfloor \theta} \geq (1 - 2\epsilon) \frac{1}{\lfloor \ell \rfloor \theta} \]

Similarly, since the value \( h_{\alpha,\ell} \) is always non-negative, it holds

\[ P(k \geq 2, u_{k-1} = v, u_k = w) \leq \frac{1}{\lfloor \ell \rfloor \theta} \]

Consider one execution of Algorithm 1 with \( k \) chosen uniformly from \([\ell]\) and let \( S \) denote the event that the execution does not end with failure. Let \( e \) be the sampled edge and \( e', e'' \) some fixed edges. Then since \( P(e = e' | S) = P(e = e')/P(S) \) and for any fixed \( e' \) it holds that

\[ (1 - \epsilon) \frac{1}{\lfloor \ell \rfloor \theta} \leq P(e' = e) \leq \frac{1}{\lfloor \ell \rfloor \theta} \]

it follows that

\[ 1 - \frac{1}{\epsilon} \leq P(e = e') = P(e = e'') \leq (1 - \frac{1}{\epsilon})^{-1} \leq 1 + \epsilon \]

where the last inequality holds because \( \epsilon \leq \frac{1}{2} \). Algorithm 1 perform sampling attempts until one succeeds. This means that the returned edge comes from the distribution conditional on \( S \). As we have noted, this scales the sampling probabilities of all edges by the same factor of \( P(S) \) and the output distribution is, therefore, pointwise \( \epsilon \)-close to uniform.

**Time complexity.** Consider again one execution of Algorithm 1. Since for every fixed edge \( e' \), the probability that \( e = e' \) is at least \( (1 - \epsilon) \frac{1}{\lfloor \ell \rfloor \theta} \), the total success probability is

\[ P(S) = P(\bigvee e = e') = \sum_{e' \in E} P(e = e') \geq m(1 - \epsilon) \frac{1}{\lfloor \ell \rfloor \theta} \]

where the second equality holds by disjointness of the events. The expected number of calls of Algorithm 1 is then

\[ \frac{\lfloor \ell \rfloor \theta}{m(1 - \epsilon) n} = \sqrt{2}(\frac{1}{1 + \epsilon}) n \geq O\left(\frac{n}{\sqrt{m}} \log \frac{1}{\epsilon}\right) \]

Each call of Algorithm 1 takes in expectation \( O(1) \) time because in each step of the random walk, we abort with probability at least \( 1/2 \) (we are on a heavy vertex as otherwise we would have aborted, in order not to abort, the next vertex also must be heavy; the fraction of neighbors that are heavy is \( h_{\alpha,1} \leq 1/2 \)). Therefore, the complexity is as claimed.

### 3.2 Biased Vertex Sampling using Hash-Ordered Access

We now describe a sampling procedure (Algorithm 4) which allows us to sample vertices such that vertices with high degree are sampled with higher probability. We will then use this for sampling multiple edges and for approximate edge counting. The procedure is broken up into two parts – one for pre-processing and one for the sampling itself. We do not make it explicit in the pseudocode how the data structures built during pre-processing are passed around for the sake of brevity.

In the rest of this section, let \( p_N = \frac{2 \log n + \log \delta^{-1}}{n} \). \( p_N \) is a parameter that determines the sampling probabilities; see Lemma 7 for the exact role this parameter plays. The algorithm works as follows. We make \( \log n \) vertex samples (line 2) which we call \( S_k \)'s. We will then have a priority queue for each \( S_k \) called \( Q_k \) which allows us to iterate over \( N(S_k) \) in the order of increasing value of \( h(v) \). This could be done by inserting all vertices in \( N(S_k) \) into the priority queue. We, however, use a more efficient way based on the hash-ordered neighbor access. We start with a priority queue that has for each vertex \( v \) from \( S_k \) its first neighbor (as that is the one with the lowest hash value) represented as \( (v, 1) \) with the priority equal to its hash. Whenever we use the pop operation, popping the \( i \)-th neighbor of \( v \in S_k \) represented as \( (v, i) \), we insert into \( Q_k \) the next neighbor of \( v \), represented as \( (v, i + 1) \) with priority equal to its hash. This allows us to access \( N(S_k) \) in the order of \( h(v) \).

To be able to perform multiple independent runs of the biased vertex sampling algorithm, we will have to resample the hash value from an appropriate distribution for any vertex that the algorithm processes. We will call these resampled hash values virtual and will denote them \( h'(v) \). At the beginning, \( h'(v) = h(v) \) for all vertices \( v \). When a vertex \( v \) from \( N(S_k) \) is processed, we put it, represented as \( (v, \text{"virtual"}) \), back into the priority queue \( Q_k \) with priority equal to the virtual hash. Each priority queue \( Q_k \) therefore contains vertices of the form \( (v, i) \), representing \( v[i] \), which have not been used by the algorithm yet and vertices of the form \( (v, \text{"virtual"}) \) that have been processed already but they have been re-inserted into the priority queue with a new priority \( h'(v) \).

Note that a vertex may be present in the priority queues multiple times (for example, if \( v \) is the \( i \)-th neighbor of \( u \) and \( j \)-th neighbor of \( w \), then \( v \) may be once inserted as \( (u, i) \) and once as \( (w, j) \) and once as \( (u, \text{"virtual")}) \)). We make sure that all copies of one vertex always have the same priority, namely \( h'(v) \). We assume in the algorithm that the hash values of all vertices are different (this happens with probability 1). One may also use \( (h'(v), id(v)) \) as the...
priority of the vertex \( v \) (with comparisons performed lexicographically) to make the algorithm also work on the event when two vertices have the same hash. This may be useful if implementing the algorithm in practice.

In the pre-processing phase, we initialize the priority queues \( Q_1, \ldots, Q_{\log n} \). These will be then used in subsequent calls of the biased vertex sampling algorithm. As we mentioned, we do not make it explicit how they are passed around (they can be thus thought of as global variables).

In what follows, we let \( v(a) = w[i] \) for \( a = (w, i) \) and \( v(a) = w \) for \( a = (w, "virtual") \). We assume that the variables \( T_k \) are sets (as opposed to multisets).

**Algorithm 3:** Biased vertex sampling – pre-processing algorithm, given a tradeoff parameter \( \theta \)

```plaintext
1 for \( k \in \{0, \ldots, \log n\} \) do
2     \( S_k \leftarrow \) sample \( \frac{n}{2^k} \) vertices with replacement
3     if \( d(S_k) \geq \frac{4m \log(2n/\delta)}{2^k \theta} \), go back to line 2
4     for \( v \in S_k \) do
5         Inserts into \( Q_k \) the tuple \((v, 1)\) with priority \( h'(v[1])\)
6 end
```

**Algorithm 4:** Biased vertex sampling – sampling algorithm, given parameters \( \theta, p_N \)

```plaintext
1 Let \( \ell \) be such that this is the \( \ell \)-th execution of the algorithm.
2 for \( k \in \{0, \ldots, \log n\} \) do
3     \( T_k \leftarrow \emptyset \)
4     while either \( h'(v(a)) \leq 1 - (1 - p_N 2^k)\ell \) or \( p_N 2^k \geq 1 \) do
5         \( a \leftarrow Q_{k}.top() \)
6         while \( v(b) = v(a) \) for \( b \leftarrow Q_{k}.pop() \) do
7             if \( b \) is of the form \((v, i)\) then
8                 Replace \((v, i)\) by \((v, i+1)\) in \( Q_k \), set priority to \( h'(v[i+1]) \)
9             end
10       end
11     if \( d(v(a)) \notin [2^k \theta, 2^{k+1} \theta] \) then
12         Skip \( a \), continue with next iteration of the loop
13       end
14     \( T_k \leftarrow T_k \cup \{v(a)\} \)
15     \( h'(v(a)) \leftarrow \text{Unif}\{[1 - (1 - p_N 2^k)\ell, 1]\} \) or 1 if \( 1 - (1 - p_N 2^k)\ell > 1 \)
16     Add to \( Q_k \) an item \((v(a), "virtual")\) with priority \( h'(v(a)) \)
17 end
18 return \( T_0 \cup \cdots \cup T_{\log n} \)
```

**Remark.** In practice, implementing Algorithm 4 poses the problem that the virtual hash values \( h'(v) \) may be quickly converging to 1 as \( \ell \) increases, making it necessary to use many bits to store them. This can be circumvented as follows. Whenever \( 1 - (1 - p_N 2^k)\ell \geq 1/2 \) during the execution of the algorithm, we access the whole neighborhood of the vertex, allowing us to resample the hashes (by setting virtual hashes) from scratch. In other words, when we have seen in expectation half of the vertices adjacent to a vertex, we get all of them, allowing us to resample them from the same distribution they had at the beginning. (Using this approach would require a minor modification to Algorithm 4.)

In what follows, let \( h'_u(v) \) be the virtual hash at the beginning of the \( \ell \)-th execution of Algorithm 4 and let \( k_\ell = \lfloor \log(d(v)/\theta) \rfloor \).

**Lemma 6.** Condition on each heavy vertex \( v \) having a neighbor in \( S_k \). At the beginning of the \( \ell \)-th execution of Algorithm 4, all heavy \( v \) with \( p_N 2^k < 1 \), \( h'(v) \) is distributed uniformly on \([1 - (1 - p_N 2^k)\ell - 1, 1]\) and the events \( h'(v) \leq 1 - (1 - p_N 2^k)\ell \) for \( v \in V \) and \( \ell \in \mathbb{Z}^+ \) are jointly independent. Moreover, any \( u \) such that \( v = u \) in \( Q_k \) has priority \( h'(u) \).

**(Proof sketch.)** We first focus on the distribution of the virtual hashes; we prove that the priorities are equal to the hashes afterwards. The proof of both parts is by induction on \( \ell \).

We define \( X_{\ell}\ell \) to be the indicator for \( h'_u(v) \leq 1 - (1 - p_N 2^k)\ell \).

We now focus on one vertex \( v \) and drop it in the subscript. We prove by induction that, conditioned on \( X_1, \ldots, X_{\ell-1} \), the distribution of \( h'_u(v) \) is uniform on \([1 - (1 - p_N 2^k)\ell - 1, 1]\). This implies that the unconditional distribution of \( h'_u(v) \) is as claimed. We will also use this below to prove independence. The distribution of \( h'_u(v) \) is as claimed as the virtual hash values are initially equal to the original (non-virtual) hash values which are assumed to be uniformly distributed on \([0, 1]\) and we are conditioning on an empty set of random variables. Consider \( h_{\ell-1}(v) \) conditioned on \( X_1, \ldots, X_{\ell-2} \). The distribution is uniform on \([1 - (1 - p_N 2^k)\ell - 2, 1]\) by the inductive hypothesis. Conditioning on \( X_{\ell-1} = 1 \), we resample \( h'_u(v) \) uniformly from \([1 - (1 - p_N 2^k)\ell - 1, 1]\). Conditioning on \( X_{\ell-1} = 0 \), it is equivalent to conditioning on \( h'_u(v) > 1 - (1 - p_N 2^k)\ell \). This conditional distribution is uniform on \([1 - (1 - p_N 2^k)\ell - 1, 1]\). Either way, the distribution is as claimed. This proves the inductive step.

Independence follows by a calculation from the fact that the hash values are independent. This part of this proof appears in the full version of this paper.

We now argue that the priorities are equal to the virtual hashes. Whenever a vertex is added to \( Q_k \), its priority is equal to its virtual hash. The only way it may happen that the priorities and virtual hashes are not equal is that the virtual hash of some vertex \( u \) changes while there exists \( a \in Q_k \) such that \( v(a) = u \). This never happens as the virtual hash of \( v(a) \) changes only on line 14 after all \( a \in Q_k \) such that \( v(a) = u \) have been removed. Note that we are using the fact that different vertices have different virtual hashes (which we assume without loss of generality as we discussed above), which ensures that all \( a \) with \( v(a) = u \) are removed on line 5.

**Lemma 7.** Let us have integer parameters \( \theta, \ell \geq 1 \). When executed \( t \) times, Algorithm 4 returns samples \( T_1, \ldots, T_t \). Assume Algorithm 4 is given the priority queues \( \{Q_k\}_{k=1}^{\log n} \) produced by Algorithm 3. Then
there is an event \( E \) with probability at least \( 1 - \delta \) such that, conditioning on this event, for any \( i \in [t] \) and any vertex \( v \) such that \( d(v) \geq \theta \), it holds that

\[
P(v \in T_i) = \min(1, p_N 2^k \frac{d(v)}{2\theta}) \in \left[ \min(1, \frac{d(v)}{2\theta} p_N), \min(1, \frac{d(v)}{2\theta} p_N) \right]
\]

and, conditionally on \( E \), the events \( \{v \in T_i\}_{i \in [t]} \) are jointly independent. Algorithm 3 has expected time complexity \( O\left(\frac{n \log n \log(n/\delta)}{\theta} \right) \).

Algorithm 4 has expected time complexity \( O\left(t \left( 1 + \frac{p_N m \log^2 n \log(n/\delta)}{\theta} \right) \right) \).

Proof. We start by specifying the event \( E \) and bounding its probability. The probability that a vertex of degree at least \( 2^k \theta \) does not have a neighbor in \( S_k \) after one sampling of \( S_k \) (that is, not considering the repetitions) on line 2 is at most

\[
1 - \frac{2^k \theta}{n} \leq \exp(-\log(2n/\delta)) = \frac{\delta}{2n}.
\]

Therefore, taking the union bound over all vertices, the probability that this happens with probability at least \( \frac{\delta}{2n} \) is satisfied in the rest of this proof, we condition on \( E \). We now prove correctness. In the rest of this proof, we condition on \( E \).

We now prove correctness. Consider a vertex \( v \) and let again \( k = \lfloor \log(d(v)/\theta) \rfloor \). Consider the case \( p_N 2^k \geq 1 \). Conditioned on \( E \), one of its neighbours is in \( S_k \). Since \( p_N 2^k \geq 1 \), the condition on line 4 is satisfied in the \( k \)-th iteration of the loop on line 2. Therefore, the whole neighborhood of \( S_k \) is added to \( T_k \) and the returned sample thus contains \( v \).

We now consider the case \( p_N 2^k < 1 \). Conditioned on \( E \), a vertex \( v \) is included in \( T_k \) when \( 2^k \theta \leq d(v) < 2^{k+1} \theta \) and \( h^k(v) \leq 1 - (1 - p_N 2^k)^{f^k-1} \). Since \( h^k(v) \) is uniformly distributed in \([1 - (1 - p_N 2^k)^{f^k-1}, 1] \), this happens with probability

\[
1 - (1 - p_N 2^k)^{f^k} - (1 - (1 - p_N 2^k)^{f^k-1}) = p_N 2^k
\]

Let \( h^k(v) \) and \( k_e \) be defined as in the statement of Lemma 6. We know from it that the events \( \{ h^k(v) \leq 1 - (1 - p_N 2^k)^{f^k} \}_{v \in V} \) are jointly independent, conditioned on \( E \). \( h^k(v) \leq 1 - (1 - p_N 2^k)^{f^k} \) is equivalent to \( v \in T_k \). This implies that the events \( \{ v \in T_k \}_{v \in V} \) are also jointly independent, as we set out to prove.

We now prove the claimed query complexity. We first show the complexity of Algorithm 3. As we argued, the probability of resampling \( S_k \) because the condition on line 3 is satisfied, is at most \( 1/2 \). Therefore, the time spent sampling the set \( S_k \) (including the repetitions) is \( O(|S_k|) \). For every \( k \), \( |S_k| = \frac{n \log(2n/\delta)}{2^k \theta} \). Therefore, the total size of the sets \( S_k \) is upper-bounded by

\[
\sum_{k=0}^{\infty} \frac{n \log(2n/\delta)}{2^k \theta} = \frac{2n \log(2n/\delta)}{\theta}.
\]

Since all values that are to be inserted into \( Q_k \) in Algorithm 3 are known in advance, we can build the priority queues in time linear with their size. This means that the preprocessing phase (Algorithm 3) takes \( O\left(\frac{2n \log(n/\delta)}{\theta} \right) \) time.

We now focus on Algorithm 4. We now prove that an execution of the loop on line 2 of Algorithm 4 takes in expectation \( O\left(1 + \frac{p_N m \log n \log(n/\delta)}{\theta} \right) \) time for any \( k \in \{0, \ldots, \log n\} \), from which the desired bound follows. Specifically, we prove that the number of executions of lines 6-8 is \( O\left(\frac{p_N m \log n \log(n/\delta)}{\theta} \right) \) from which this bound follows since every iteration takes \( O(\log n) \) time (as the time complexity of an iteration is dominated by the operations of the priority queue).

Lines 6-8 are executed once for each item \( a \) in \( Q_k \) such that the priority of \( a \) is \( 1 - (1 - p_N 2^k)^f \) if \( p_N 2^k < 1 \) or when \( p_N 2^k \geq 1 \). As we have argued, this happens with probability \( \min(1, p_N 2^k) \leq p_N 2^k \). There are \( \frac{n \log(2n/\delta)}{2^k \theta} \) vertices in \( S_k \). Since these vertices are chosen at random, there is in expectation \( \frac{2m \log(2n/\delta)}{2^k \theta} \) incident edges. We consider an item for such incident edge with probability \( \leq p_N 2^k \). This means that we consider on the mentioned lines in expectation \( O\left(\frac{p_N m \log(2n/\delta)}{\theta} \right) \) edges, as we wanted to prove. \( \square \)

3.3 Bernoulli Sampling with Hash-Ordered Neighbor Access

We now show how to sample each edge independently with some fixed probability \( p \) in the hash-ordered neighbor access model. Our approach works by separately sampling light and heavy edges, then taking union of the samples. In fact, we solve a more general problem of making \( t \) Bernoulli samples with time complexity sublinear in \( t \) for some range of parameters. We will need this for sampling edges with replacement. We first give an algorithm to sample edges, assuming we can sample separately light and heavy edges.

Theorem 8. Given a parameter \( t \), with probability at least \( 1 - \delta \), Algorithm 5 returns \( t \) samples \( T_1, \ldots, T_t \). Each edge is included in \( T_i \) with probability \( p \). Furthermore, the events \( \{ e \in T_i \}_{i \in [t]} \) are jointly independent. The expected time complexity is \( O\left(\sqrt{p m \log(n)} \log(n/\delta)^4 + t(1 + p m \log^2 n \log(n/\delta))\right) \).

Proof. If \( p > 0.9 \), we read the entire graph in \( O(n + m) \) and compute the sample in time \( O(t p m) \). This is less than the claimed complexity. In the rest, we assume that \( p \leq 0.9 \).

Correctness follows from lemmas 9 and 10 which imply that light and heavy edges, respectively, are separately sampled independently with the right probability. Moreover, there is no dependency between the samples of the light and heavy edges, as the sample of light edges does not depend on the hashes. The same lemmas give running times of \( O(tp m \theta) \) and \( O(tp m \log^2 n \log(n/\delta)) \) spent on
Algorithm 5: Make \( t \) Bernoulli samples from \( E \) with inclusion probability \( p \)

1. if \( p > 0.9 \) then
2. Use standard \( O(n^2 \log n) \) algorithm
3. end
4. \( \theta \leftarrow \sqrt{\frac{\log(n) \log(n/\delta)}{p}} \)
5. Run Algorithm 3 with parameter \( \theta \) to prepare data structures for Algorithm 4 (used within Algorithm 7)
6. for \( i \) from 1 to \( t \) do
7. \( S_{L,i} \leftarrow \) sample each light edge with probability \( p \) using Algorithm 6 with parameter \( \theta \)
8. \( S_{H,i} \leftarrow \) sample each heavy edge with probability \( p \) using Algorithm 7 with parameter \( \theta \)
9. end
10. return \((S_{L,1} \cup S_{H,1}, \ldots, S_{L,t} \cup S_{H,t})\)

lines 7 and 8 respectively. Algorithm 3 takes \( \frac{n \log n \log(n/\delta)}{\theta} \) time. Substituting for \( \theta \), the expected time complexity is as claimed. \( \Box \)

Sampling light edges. Now we show how to sample from the set of light edges such that each light edge is sampled independently with some specified probability \( p \).

Algorithm 6: Sample each light edge independently with probability \( p \)

1. \( k \sim \text{Bin}(n\theta, p) \)
2. \( T \leftarrow \emptyset \)
3. \( M \leftarrow \emptyset \)
4. repeat \( k \) times
5. \( v \leftarrow \text{pick vertex uniformly at random} \)
6. \( \ell \leftarrow \text{random number from } [\theta] \)
7. if \((v, \ell) \in M\) then
8. Go to line 5.
9. end
10. if \( v \) is light and \( d(v) \geq \ell \), add the \( \ell \)-th edge incident to \( v \) to \( T \)
11. \( M \leftarrow M \cup \{(v, \ell)\} \)
12. end
13. return \( T \)

As essentially the same algorithm already appeared in [14], we defer the proof of the following lemma to the full version.

**Lemma 9.** Algorithm 6 samples each light edge independently with probability \( p \leq 0.9 \) and its expected query complexity is \( O(n \theta) \).

Sampling heavy edges. We now show an algorithm for Bernoulli sampling from the set of heavy edges. The algorithm is based on Algorithm 4.

**Lemma 10.** Assume that Algorithm 7 is given \( \{Q_k\}_{k=1}^{n} \) as set by Algorithm 3. With probability at least \( 1 - \delta \), Algorithm 7 samples each heavy edge independently with probability \( p \). Moreover, when executed multiple times, the outputs are independent. It has time complexity \( O(pm \log^2 n \log(n/\delta)) \) with high probability.

Algorithm 7: Sample each heavy edge independently with probability \( p \), given parameter \( \theta \)

1. \( S \leftarrow \) Use Algorithm 4 with \( p_{N} = \min(1, 2\theta p) \)
2. \( S' \leftarrow \) heavy vertices from \( S \)
3. \( T \leftarrow \emptyset \)
4. for \( v \in S' \) do
5. With probability \( \frac{1 - (1 - p)^{d(v)}}{\min(1, p_{N}2^{\log d(v)/\theta})} \), skip \( v \) and continue
6. \( k \sim \text{Bin}(d(v), p)|k \geq 1) \)
7. Sample \( k \) edges incident to \( v \) without replacement, add them to \( T \)
8. end
9. return \( T \)

**Proof.** We first show that the probability on line 4 is between 0 and 1 (otherwise, the algorithm would not be valid). It is clearly non-negative, so it remains to show \( 1 - (1 - p)^{d(v)} \leq \min(1, p_{N}2^{\log d(v)/\theta}) \).

If \( 1 \leq p_{N} = 2\theta p \), then \( 1 \leq p_{N}2^{\log d(v)/\theta} \), and the inequality then clearly holds. Otherwise, \( p_{N}2^{\log d(v)/\theta} \geq 2\theta p 2^{\log d(v)/\theta} - 1 \geq pd(v) \geq 1 - (1 - p)^{d(v)} \).

*We now show correctness.* That is, we show that edges are sampled independently with the desired probabilities. By Lemma 7, for heavy \( v \), it holds that \( P(v \in S) = \min(1, p_{N}2^{\log d(v)/\theta}) \). This means that the probability of \( v \) being in \( S \) and not being skipped is \( \min(1, p_{N}2^{\log d(v)/\theta}) \frac{1 - (1 - p)^{d(v)}}{\min(1, p_{N}2^{\log d(v)/\theta})} = 1 - (1 - p)^{d(v)} \).

This is equal to the probability of \( X \geq 1 \) for \( X \sim \text{Bin}(d(v), p) \). Since \( k \) is picked from \( \text{Bin}(d(v), p)|k \geq 1) \), the distribution of number of edges incident to \( v \) that the algorithm picks is distributed as \( \text{Bin}(d(v), p) \). Let \( X_{o} \) be the number of edges incident to \( v \) that are picked. Since each vertex \( v \) is picked independently by Lemma 7, we have that these random variables are independent and \( X_{o} \sim \text{Bin}(d(v), p) \). Consider the vector \( (X_{o}, \ldots, X_{o}) \). Consider experiment where we pick each edge independently with probability \( p \) (this is the desired distribution) and let \( Y_{o} \) be the number of edges incident to \( v \) that are picked. Then \( (Y_{o}, \ldots, Y_{o}) \sim (X_{o}, \ldots, X_{o}) \). By the conditioning principle, we get that each (directed) edge is sampled independently with probability \( p \).

*Assume Algorithm 7 is executed \( t \) times, outputting \( H_{1}, \ldots, H_{t} \). By Lemma 7, we know that the events \( \{e \in H_{i}\}_{i=1}^{t} \) are jointly independent. Algorithm 7 only depends on the virtual hashes in the calls of Algorithm 4 and the rest only depends on independent randomness. The samples \( H_{1}, \ldots, H_{t} \) are thus independent.*

*We now argue the time complexity.* Algorithm 4 has expected time complexity \( O(Nm \log^2 n \log(n/\delta)) = O(pm \log^2 n \log(n/\delta)) \). We now prove this dominates the complexity of the algorithm. The rest of the algorithm has time complexity linear in \(|S| + |T| \). The complexity of Algorithm 4 clearly dominates \(|S| \) (as \( S \) is the output of this algorithm). It holds \( E(|T|) = pm \), so the expected size of \( T \) is also dominated by the complexity of Algorithm 4. \( \Box \)
3.4 Sampling Edges with Replacement with Hash-Ordered Neighbor Access

The algorithm for Bernoulli sampling can be used to sample multiple edges with replacement. The proof of the theorem below appears in the full version. We now sketch intuition of correctness of the algorithm. Suppose $S_1$ is non-empty. Picking from each non-empty $S_i$ one edge at random gives us a sample without replacement of size equal to the number of non-empty $S_i$’s. Since the number of $S_i$’s is $2s$, if we pick $p$ large enough, then with high probability at least $s$ of them will be non-empty. In that case, we have a sample without replacement of size $\geq s$ and taking a random subset of size $s$ gives a sample with the desired distribution.

Algorithm 8: Sample $s$ edges with replacement

1. $p \leftarrow 1/n^2$
2. $S_1, \ldots, S_2s \leftarrow$ Bernoulli samples with $\min(1, p)$ using Algorithm 5 with failure probability $\frac{p}{3} \epsilon n^2$
3. $T \leftarrow$ from each non-empty $S_i$ pick a random edge
4. if $|T| < s$ then
5. $p \leftarrow 2p$
6. Go to line 2
7. end
8. return random subset of $T$ of size $s$

Theorem 11. Given $s \leq n$, with probability at least $1 - \delta$, Algorithm 8 returns $s$ edges sampled with replacement. Moreover, the algorithm runs in time $O(\sqrt{s} \log n \log(n/\delta) + s \log^2 n \log(n/\delta))$ with high probability.

ACKNOWLEDGMENTS

The authors are part of BARC, supported by the VILLUM Foundation grant 16582.

REFERENCES


