This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2016.2556663, IEEE Transactions on Knowledge and Data Engineering

Counting Triangles in Large Graphs by Random Sampling

Bin Wu, Ke Yi, and Zhenguo Li

Abstract—The problem of counting triangles in graphs has been well studied in the literature. However, all existing algorithms, exact or approximate, spend at least linear time in the size of the graph (except a recent theoretical result), which can be prohibitive on today's large graphs. Nevertheless, we observe that the ideas in many existing triangle counting algorithms can be coupled with random sampling to yield potentially sublinear-time algorithms that return an approximation of the triangle count without looking at the whole graph. This paper makes these random sampling algorithms more explicit, and presents an experimental and analytical comparison of different approaches, identifying the best performers among a number of candidates.

Index Terms—Triangle counting, random sampling

1 Introduction

Graphs are a ubiquitous form to represent and model complex relationships between entities in various fields, including biochemistry, information systems, and social networks. Triangle is one of the most fundamental substructures of a graph. In social network analysis, two fundamental measurements, the *clustering coefficient* [1] and the *triangle connectivity* [2], are both derived from the number of triangles. Various applications depend on triangle listing and counting, such as uncovering hidden thematic structures [3], detecting Web spam [4], and community detection [5].

The problems of both listing and counting (exactly or approximately) all triangles in a given graph have been extensively studied in the literature, from as early as a 1977 STOC paper [6] to the 2013 SIGMOD best paper [7]. However, all existing algorithms, exact or approximate, spend at least linear time, visiting each vertex and edge of the graph at least once (except a recent theoretical result [8]). The motivation of our study is that sublinear time is actually possible to obtain a good estimate of triangle count, and this is important in a number of scenarios. First, as today's graphs easily contain billions of vertices and edges, even linear time can be prohibitive. Second, as the number of triangles is often used in analyzing some statistical properties of the graph, very often we do not need an exact answer. An approximation (say, within 10% of the true count) would be just as good. Third, when the graph is dynamically changing (i.e., insertion/deletion of vertices and edges), and we would like to count the triangles periodically so as to monitor the dynamics of the graph, using a linear-time algorithm to do the counting every time would be too expensive and compromises the timeliness of the monitoring.

Nevertheless, although prior work has not explicitly considered the sublinear-time triangle counting problem, we observe that the ideas in many existing linear or super-linear algorithms can actually be coupled with random sampling to make the algorithm potentially sublinear-time. In this paper, we make this connection clearer, and more importantly, provide a detailed analytical and experimental comparison of different random sampling strategies to the approximate triangle counting problem, identifying the best performers among a number of candidates.

- *B. Wu and K. Yi are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology Clear Water Bay, Hong Kong, China. Email: {bwuac,yike}@cse.ust.hk.*
- *Z. Li is with Huawei Noah's Ark Lab, Hong Kong, China. Email: li.zhenguo@huawei.com.*

2 Prior work

We classify the existing algorithms into those that count the number of triangles exactly and those that do so approximately.

2.1 Exact counting algorithms

Most exact counting algorithms actually solve the *listing* problem, i.e., they enumerate all the triangles in the graph, thus obtain the triangle count as a by-product [6], [9], [10]. These algorithms run in $O(n^3)$ or $O(m^{1.5})$ time. Here, n denotes the number of vertices and m the number of edges. This running time is optimal in the worst case since there can be as many as $O(n^3)$ or $O(m^{1.5})$ triangles in the graph. There are also algorithms that count the triangles without listing them by using matrix multiplication [6]. They have running time $O(n^{2.37})$, which is better than the listing algorithms but only if the graph is dense enough. There is an extensive experimental study on the performance of these exact counting and listing algorithms [10]. Recently, as the graphs get even larger, it also has attracted a lot of interests to extend the these algorithms to the external memory model [7], [11], [12], [13], [14], [15] and the MapReduce model [16], [17], [18], [19], [20].

2.2 Approximate counting algorithms

In view of the high running times of the exact counting algorithms and the fact that an approximate count satisfies the need of most applications, faster approximation algorithms have been sought for. The easiest method is to sample a small subgraph from the whole graph, count the number of triangles in the subgraph, and scale up the count. In particular, *Doulion* [21] forms the subgraph by sampling each edge with probability p. Since each triangle in the original graph appears in the sampled subgraph with probability p^3, the number of triangles in the subgraph is multiplied by $1/p^3$ to get an unbiased estimator of the true count. Flipping a coin with probability p

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2016.2556663, IEEE Transactions on Knowledge and Data Engineering

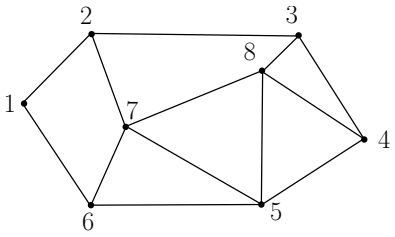


Fig. 1. An example graph

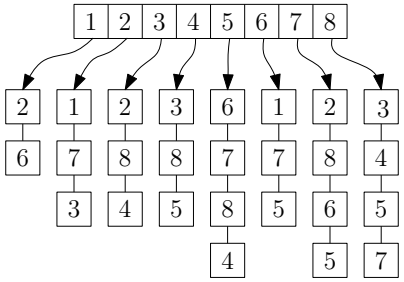


Fig. 2. The adjacency list representation

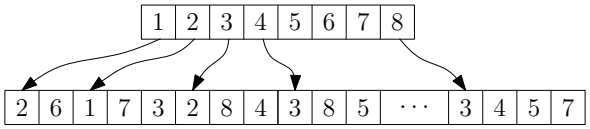


Fig. 3. The edge array representation

for each edge in the graph requires at least a linear scan of the whole graph, so this is not a sublinear-time algorithm, strictly speaking. Nevertheless, this can be avoided by sampling the edges with (or without) replacement. We describe this version of *Doulion* more explicitly in Section 4.

Another sampling method proposed in the literature is *wedge sampling*. A *wedge* is any length-2 path in the graph, thus a triangle is formed when a wedge is closed. Assuming that a wedge can be randomly sampled from the graph, then a sublinear-time algorithm can be obtained [22]. However, common graph representations (e.g., adjacency list or adjacency matrix) do not support wedge sampling. To support this operation, the graph has to be preprocessed in $O(n)$ time, and the preprocessing needs to be done again every time the graph has changed.

The problem has also received a lot of attention in the streaming model [23], [24], [25], [26], [27], [28], [29], [30], and all streaming algorithms return approximate triangle counts (exact counting is known to be impossible in the streaming model). Since any streaming algorithm makes at least one pass over the whole input, they do not yield sublinear-time algorithms. Nevertheless, some of the streaming algorithms, such as the one in [28], can be modified to run in sublinear time, and we describe this modification in more detail in Section 4.

Very recently, there have been some theoretical studies on approximating the number of triangles in sublinear time [8], [31]. However, those algorithms are far from practical, according to our experimental evaluation.

3 Preliminaries

Let $G = (V, E)$ be a simple undirected graph with n vertices and m edges. A *triangle* in G is a triple of 3 vertices (u, v, w) such that $\{(u, v), (v, w), (u, w)\} \subseteq E$. We denote by $\Delta(G)$ the set of all triangles in G, and denote the triangle count as $T_3 = |\Delta(G)|$. For any $v \in V$, let $N(v) = \{u \in V \mid (u, v) \in E\}$ denote the *neighbors* of v. The *degree* of v is $d(v) = |N(v)|$. For any vertex $v \in V$, let $\lambda(v)$ be the number of triangles having v as one of the vertices; similarly for any edge $e \in E$, $\lambda(e)$ denotes the number of triangles having e as one of the edges. Note that for any $e = (u, v)$, $\lambda(e) = |N(v) \cap N(u)|$. We assume that each vertex has a unique integer id.

Example: For the example graph in Figure 1, we have $\lambda(1) = 0, \lambda(8) = 3$ (triangles $(7, 8, 5), (4, 5, 8)$ and $(3, 4, 8)$ have 8 as a vertex), $\lambda(5, 8) = 2$ (triangles $(5, 7, 8)$ and $(4, 5, 8)$ have $(5, 8)$ as an edge).

We assume the adjacency list representation for the graph, the most commonly used storage format for graphs. But depending on whether the graph is static or dynamic, there can be two different implementations. The first one is the "textbook" method, which uses an array indexed by the vertices. (This assumes that the vertices are numbered from 1 to n. If the vertex id's are arbitrary, another level of indirection is needed to map the vertex id's to numbers from 1 to n using a hash table.) Each cell in the vertex array points to a linked list that stores all the neighbors of that vertex. For example, the adjacency list representation of the graph in Figure 1 is shown in Figure 2. We assume that each neighbor list maintains the size of the list, which is equal to the degree of the corresponding vertex. Such a representation easily supports dynamic changes to the graph, i.e., inserting or deleting an edge.

The other implementation simply concatenates all the neighbor lists into one big array, which we call the *edge array*. Each pointer in the vertex array now points to the first neighbor of the corresponding vertex in the edge array (Figure 3). This implementation does not easily support changes to the graph, but it is more compact. In particular, since we no longer need the pointers inside each neighbor list, the space usage is reduced to half (or $1/3$ if doubly-linked lists are used in the adjacency list representation). Furthermore, for each vertex, all its neighbors are stored consecutively in memory (or on disk), so traversal of its neighbors is much more cache-efficient than in the adjacency list representation. Thus, this is the preferred storage format for graphs which see no (or few) changes. Also note that in this representation, we no longer need to store the degree for each vertex v_i explicitly, as it can be computed from the starting address of $N(v_i)$ and that of $N(v_{i+1})$.

When it comes to random sampling algorithms, these two representations allows different sampling strategies to explore the graph. Both representations allow us to uniformly sample a vertex. After a vertex v is sampled, we can retrieve all its neighbors in $O(d(v))$ time. The edge array representation also allows us to sample a neighbor of a specific vertex more efficiently, as well as sample an edge uniformly from the edge array. This turns out to be an important operation that can lead to more accurate estimation of the triangle count.

Finally, all algorithms presented in this paper report an unbiased estimator of T_3 with each sampling step. Then we simply take the average of multiple estimators to improve the accuracy. This immediately brings two benefits: (1) The algorithm itself "knows" how well it has been doing: One can use standard statistical formulas to estimate the standard

deviation and confidence intervals of the result from these estimates, and stop the algorithm when the accuracy is good enough; (2) The algorithm is "embarrassingly parallel", and can be easily implemented in a parallel/distributed graph management system like Pregel [32] or GraphLab [33].

4 Algorithms

4.1 Subgraph sampling

As mentioned in Section 2, *Doulion* [21] samples a subgraph from G by picking every edge with probability p, counts the number of triangles in the sampled subgraph, and then scales up the count by a factor of $1/p^3$. It has been shown [21] that this algorithm returns an unbiased estimator with variance $\frac{T_3(p^3-p^6)+2s(p^5-p^6)}{p^6}$ where s is the number of pairs of triangles that share a common edge.

This algorithm, as stated, is not a sublinear-time algorithm as it flips a coin for every edge of the graph. Nevertheless, we can replace this coin-flip sampling by sampling without replacement. More precisely, we randomly pick a subset of k edges from all m edges to form the subgraph. Note that the probability of a triangle appearing in the subgraph under this sampling method is $p' = \binom{k}{3}/\binom{m}{3}$, so we scale up the triangle count by $1/p'$.

The variance of the estimator under this sampling method is hard to compute exactly, because in sampling without replacement, the edges are not independently picked. But the leading term, T_3/p' is still correct, which we use as a good approximation of the actual variance.

This algorithm only works in the edge array model, because it needs to sample the edges uniformly at random.

4.2 Vertex sampling

The basic idea of *vertex sampling* roots from an exact triangle counting algorithm called *vertex iterator* [6]. Recall that $\lambda(v)$ is the number of triangles that contain vertex v. To count all triangles, the vertex iterator algorithm simply counts $\lambda(v)$ for each v, and adds them up. Since each triangle is counted three times, the final sum is divided by 3 to obtain T_3, i.e., $T_3 = \frac{1}{3}\sum_v \lambda(v)$.

To turn this idea into a sublinear-time algorithm, we randomly sample a vertex v and compute $\lambda(v)$. Then we scale it up by a factor of $n/3$, which will be an unbiased estimator of T_3. We do this multiple times, and take the average.

To compute $\lambda(v)$, we first build a hash table on $N(v)$. Then for each $v_i \in N(v)$, we compute $|N(v_i) \cap N(v)|$ by probing the hash table on $N(v)$ with each $u \in N(v_i)$. The detailed algorithm (for one sampling step) is given in Algorithm 1. The algorithm returns $tn/6$ in the end because each triangle having v as a vertex is counted twice when summing up $|N(v_i) \cap N(v)|$ over all neighbors v_i of v.

Algorithm 1: Vertex sampling

```
t ← 0;
sample a vertex v uniformly from V;
build a hash table on N(v);
foreach v_i ∈ N(v) do
    t ← t + |N(v) ∩ N(v_i)|;
report tn/6;
```

In practice, when $N(v)$ and $N(v_i)$ are small enough, it is actually faster to sort and merge them to compute $|N(v) \cap N(v_i)|$ (i.e., a "sort-merge join"), instead of using a hash table (i.e., "hash join"). In our implementation, we use the "hash join" method when the adjacency list is long enough (≥ 700), and use the "sort-merge join" method when the lists are shorter. It should be clear that this algorithm works in both the adjacency list and the edge array model.

4.2.1 Analysis

It is straightforward to see that the output is always an unbiased estimator of T_3: When v is chosen uniformly at random, we have

$$\mathsf{E}_v[\lambda(v)] = \sum_{v \in V} \lambda(v) \cdot \frac{1}{n} = \frac{3T_3}{n}.$$

Thus, $n\lambda(v)/3$ is an unbiased estimator of T_3.

The variance of the estimator is just the variance of $\lambda(v)$ times $n^2/9$. The variance of $\lambda(v)$ is

$$\begin{aligned}\mathsf{Var}_v[\lambda(v)] &= \mathsf{E}_v[\lambda(v)^2] - \mathsf{E}_v[\lambda(v)]^2 \\ &= \sum_v \lambda(v)^2/n - (3T_3/n)^2.\end{aligned}$$

The running time of the algorithm per sampling step is (the big-Oh of)

$$d(v) + \sum_{v_i \in N(v)} d(v_i) = \sum_{v_i \in N(v)} d(v_i).$$

Since v is randomly chosen from all vertices, the expected running time is

$$\frac{1}{n}\sum_v \sum_{v_i \in N(v)} d(v_i) = \frac{1}{n}\sum_v d(v)^2.$$

Proposition 1. *The vertex sampling algorithm returns an unbiased estimator of T_3 with variance $\frac{n}{9}\sum_v \lambda(v)^2 - T_3^2$. Its running time per sampling step is $O(\frac{1}{n}\sum_v d(v)^2)$.*

4.3 Edge sampling

The *edge sampling* method is motivated by another exact triangle counting algorithm called *edge iterator* [6]. It is based on the observation that T_3 is also equal to $\frac{1}{3}\sum_e \lambda(e)$, where $\lambda(e)$ is the number of triangles that have e as an edge. Thus, if we uniformly randomly sample an edge e and compute $\lambda(e)$, then $\frac{m}{3}\lambda(e)$ will be an unbiased estimator of T_3.

It remains to describe how to sample an edge uniformly from all edges of the graph. In the edge array, this can be done by just randomly picking a location in the edge array. Note that the edge array has size $2m$ and each edge appears exactly twice in the array, so each edge is sampled with probability $1/m$. However, this only gives us one endpoint of the edge (see Figure 3). We could have required the edge array to store both endpoints, but that doubles its size, losing the benefit of compactness. The trick is to sample again. More precisely, we first uniformly sample a location in the edge array, getting a vertex u. Then we look up u in the vertex array, and uniformly sample a neighbor of u at random, denoted as v. From the vertex array, we can get the starting and the ending position of u's neighbor list in the edge array, so uniformly sampling a neighbor of u is easy. Then we return $e = (u, v)$ as the sampled edge. It is easy to see that this process will sample

every e with probability $1/m$: For any edge $e = (u, v)$, the probability that u is sampled in the first step is $d(u)/2m$, then the probability that v is sampled in the second step is $1/d(u)$, so u, v are sampled in this order with probability $1/2m$. The edge can also be obtained if v is sampled first and u second, which also happens with probability $1/2m$. Thus the overall probability that $e = (u, v)$ is sampled is $1/m$, as desired.

To compute $\lambda(e) = |N(v) \cap N(v)|$, we similarly as before do either a hash join or a sort-merge join depending on the sizes of $N(v)$ and $N(u)$. The detailed algorithm is given in Algorithm 2.

Algorithm 2: Edge sampling (edge array model)

sample a vertex u uniformly from the edge array;
sample a vertex v uniformly from $N(u)$;
$\lambda(e) \leftarrow |N(u) \cap N(v)|$;
report $\lambda(e) \cdot m/3$;

4.3.1 Analysis

By the fact that $\sum_{e \in E} \lambda(e) = 3T_3$, we can easily see that the algorithm returns an unbiased estimator for T_3. When e is chosen uniformly at random, we have

$$\mathsf{E}_e[\lambda(e)] = \sum_{e \in E} \lambda(e) \frac{1}{m} = \frac{3T_3}{m}$$

Thus, $\lambda(e) \cdot m/3$ is an unbiased estimator of T_3.

The variance of the estimator is $m^2/9k \cdot \mathsf{Var}[\lambda(e)]$, where

$$\begin{aligned}\mathsf{Var}_e[\lambda(e)] &= \mathsf{E}_e[\lambda(e)^2] - \mathsf{E}_e[\lambda(e)]^2 \\ &= \sum_e \lambda(e)^2/m - (3T_3/m)^2.\end{aligned}$$

So the variance of the estimator is $\frac{m}{9} \sum_{e \in E} \lambda(e)^2 - T_3^2$.

The running time of the algorithm is $O(d(u) + d(v))$, dominated by computing $|N(u) \cap N(v)|$. Since the edge $e = (u, v)$ is randomly chosen from all edges, the expected running times is

$$\frac{1}{m} \sum_e d(u) + d(v) = \frac{1}{m} \sum_v d(v)^2.$$

Proposition 2. *The edge sampling algorithm in the edge array model returns an unbiased estimator of T_3 with variance $\frac{m}{9} \sum_e \lambda(e)^2 - T_3^2$. Its running time per sampling step is $O(\frac{1}{m} \sum_v d(v)^2)$.*

4.3.2 Adaptation to the adjacency list model

In the adjacency list model, we cannot sample an edge uniformly. But we can still apply the same idea of edge sampling, except that we need to take the non-uniformity of sampling into account to remove the bias. The modified algorithm is shown in Algorithm 3.

Algorithm 3: Edge sampling (adjacency list model)

sample a vertex u uniformly from the vertex array;
sample a vertex v uniformly from $N(u)$;
$\lambda(e) \leftarrow |N(u) \cap N(v)|$;
report $\lambda(e) \cdot \frac{nd(u)d(v)}{3(d(u)+d(v))}$;

4.3.3 Analysis

Let $\lambda'(e) = \frac{\lambda(e)d(u)d(v)}{d(u)+d(v)}$. Its expectation is

$$\mathsf{E}_e[\lambda'(e)] = \sum_{e^*=(u,v)} \frac{\lambda(e)d(u)d(v)}{d(u) + d(v)} \Pr[e = e^*].$$

Given the sampling process in the algorithm, a particular edge $e^* = (u, v)$ is sampled if u is sampled in the first step (which happens with probability $1/n$) and v is sampled in the second step (which happens with probability $1/d(u)$, or the other around. So the probability that e^* is sampled is $\frac{1}{n}(\frac{1}{d(u)} + \frac{1}{d(v)})$. Thus, the expectation is

$$\mathsf{E}[\lambda'(e)] = \sum_e \frac{\lambda(e)}{n} = \frac{3T_3}{n}.$$

So $\lambda'(e) \cdot n/3$ is a unbiased estimator of T_3.

The variance of $\lambda'(e)$ is:

$$\begin{aligned}\mathsf{Var}[\lambda'(e)] &= \mathsf{E}[\lambda'(e)^2] - \mathsf{E}[\lambda'(e)]^2 \\ &= \sum_e \frac{\lambda(e)^2 d(u)d(v)}{n(d(u) + d(v))} - \frac{9T_3^2}{n^2}\end{aligned}$$

Note that the first sampling step takes $O(1)$ time, but the second step for sampling v takes $O(d(u))$ time as we have to traverse the neighbor list of u. But this is dominated by the time to compute $|N(u) \cap N(v)|$, which takes time $O(d(u) + d(v))$. As edge $e = (u, v)$ is sampled with probability $\frac{1}{n}(\frac{1}{d(u)} + \frac{1}{d(v)})$, the expected running time of this algorithm is

$$\begin{aligned}&\sum_{e=(u,v)} (d(u) + d(v)) \frac{1}{n} \left(\frac{1}{d(u)} + \frac{1}{d(v)} \right) \\ &= \frac{1}{n} \sum_e \frac{(d(u) + d(v))^2}{d(u)d(v)}\end{aligned}$$

Proposition 3. *The edge sampling algorithm in the adjacency list model returns an unbiased estimator of T_3 with variance $\frac{n}{9} \sum_{e \in E} \frac{\lambda(e)^2 d(u)d(v)}{d(u)+d(v)} - T_3^2$. Its running time per sampling step is $O(\frac{1}{n} \sum_e \frac{(d(u)+d(v))^2}{d(u)d(v)})$.*

4.4 Triangle sampling

The *triangle sampling* method is based on a recent triangle counting algorithm in the streaming model [28]. It can also be considered as combining the MinHash [34] idea with edge sampling: After sampling an edge e, instead of computing $|N(u) \cap N(v)|$ exactly, we try to estimate it by randomly sampling a vertex from $N(u)$ and $N(v)$ and checking if it is inside $N(u) \cap N(v)$.

We first sample an edge $e = (u, v)$ uniformly as in the edge sampling algorithm (edge array model). Then we uniformly sample a vertex neighboring to either u or v and check whether it is also a neighbor of the other. More precisely, we generate a random number i from 1 to $d(u)+d(v)$. If $1 \le i \le d(u)$, we pick the i-th neighbor of u and check whether it is also a neighbor of v; if $i > d(u)$, we pick the $(i - d(u))$-th neighbor of v and check whether it is also a neighbor of u. If the answer is yes, we have found a triangle (hence the name triangle sampling). Finally, proper scaling has to be done to turn this into an unbiased estimator of T_3. Please see the details in Algorithm 4.

Algorithm 4: Triangle sampling (edge array model)

sample a vertex u uniformly from the edge array;
sample a vertex v uniformly from $N(u)$;
generate a random number i from 1 to $d(u)+d(v)$;
$t \leftarrow 0$;
if $i \leq d(u)$ **then**
  $w \leftarrow u$'s i-th neighbor;
  if $w \in N(v)$ **then** $t \leftarrow 1$;
else
  $w \leftarrow v$'s $(i-d(u))$-th neighbor;
  if $w \in N(u)$ **then** $t \leftarrow 1$;
report $t \cdot (d(u)+d(v))m/6$;

4.4.1 Analysis

Conditioned upon $e=(u,v)$ being sampled, $t=1$ with probability

$$\frac{2|N(u)\cap N(v)|}{d(u)+d(v)} = \frac{2\lambda(e)}{d(u)+d(v)},$$

and 0 otherwise.

Let $X = t\cdot(d(u)+d(v))/2$. Since every edge $e=(u,v)$ is sampled with probability $1/m$, we have

$$\begin{aligned}\mathsf{E}[X] &= \sum_{e\in E} \frac{d(u)+d(v)}{2}\frac{2\lambda(e)}{m(d(u)+d(v))}\\ &= \sum_{e\in E}\frac{\lambda(e)}{m}\\ &= 3T_3/m.\end{aligned}$$

Thus, $X\cdot m/3$ is an unbiased estimator of T_3.

The variance of X is

$$\begin{aligned}\mathsf{Var}[X] &= \mathsf{E}[X^2]-\mathsf{E}[X]^2\\ &= \sum_e \frac{\lambda(e)(d(u)+d(v))}{2m} - \frac{9T_3^2}{m^2}\end{aligned}$$

The running time of the algorithm is dominated by checking whether w is in $N(u)$ or $N(v)$, which takes time $O(d(u))$ or $O(d(v))$. We switch to the former case with probability $d(v)/(d(u)+d(v))$, and the latter with probability $d(u)/(d(u)+d(v))$, so the expected running time is $O\left(\frac{d(u)d(v)}{d(u)+d(v)}\right)$. Since the edge is chosen uniformly at random, the overall expected running time is $O\left(\frac{1}{m}\sum_e \frac{d(u)d(v)}{d(u)+d(v)}\right)$.

Proposition 4. *The triangle sampling algorithm in the edge array model returns an unbiased estimator of T_3 with variance $\frac{m}{18}\sum_e \lambda(e)(d(u)+d(v)) - T_3^2$. Its running time per sampling step is $O\left(\frac{1}{m}\sum_e \frac{d(u)d(v)}{d(u)+d(v)}\right)$.*

4.4.2 Adaptation to the adjacency list model

As with the edge sampling algorithm, we can similarly adapt the triangle sampling algorithm to the adjacency list model. Again we will not be able to sample an edge uniformly, but still an unbiased estimator of T_3 can be obtained if the sampling probability is properly accounted for. The details are presented in the Algorithm 5.

Algorithm 5: Triangle Sampling

sample a vertex u uniformly from the vertex array;
sample a vertex v uniformly from $N(u)$;
generate a random number i from 1 to $d(u)+d(v)$;
$t \leftarrow 0$;
if $i \geq d(u)$ **then**
  $w \leftarrow u$'s i-th neighbor;
  if $w \in N(v)$ **then** $t \leftarrow 1$;
else
  $w \leftarrow v$'s $(i-d(u))$-th neighbor;
  if $w \in N(u)$ **then**
    $t \leftarrow 1$
report $t \cdot d(u)d(v)n/6$;

4.4.3 Analysis

Conditioned upon $e=(u,v)$ being sampled, we still have $t=1$ with probability $2\lambda(e)/(d(u)+d(v))$. However, the probability that $e=(u,v)$ is no longer $1/m$, but $\frac{1}{n}(\frac{1}{d(u)}+\frac{1}{d(v)})$ as we derived previously.

Let $X = t\cdot d(u)d(v)/2$. We have

$$\begin{aligned}\mathsf{E}[X] &= \sum_e \frac{d(u)d(v)}{2}\cdot\frac{1}{n}\left(\frac{1}{d(u)}+\frac{1}{d(v)}\right)\\ &\quad\cdot\frac{2\lambda(e)}{d(u)+d(v)}\\ &= \sum_e \frac{\lambda(e)}{n} = \frac{3T_3}{n}.\end{aligned}$$

Thus, $X\cdot n/3$ is an unbiased estimator of T_3.

The variance of X is

$$\begin{aligned}\mathsf{Var}[X] &= \mathsf{E}[X^2]-\mathsf{E}[X]^2\\ &= \sum_e \frac{d(u)d(v)\lambda(e)}{2n} - \frac{9T_3^2}{n^2}.\end{aligned}$$

Since in the adjacency list model, we can only scan the neighbor list of u to get its i-th neighbor in $O(1)$ time, the running time of the algorithm is $O(d(u)+d(v))$ for a given edge $e=(u,v)$. Thus, the expected running time of this algorithm is the same as that of edge sampling in the adjacency list model.

Proposition 5. *The triangle sampling algorithm returns an unbiased estimator of T_3 with variance $\frac{n}{18}\sum_e \lambda(e)d(u)d(v) - T_3^2$. Its running time per sampling step is $O(\frac{1}{n}\sum_e \frac{(d(u)+d(v))^2}{d(u)d(v)})$.*

4.5 Wedge sampling

The technique of *wedge sampling* is a new approach to approximating the triangle count [22]. A *wedge* is any length-2 path (u-v-w) in the graph. The observation is that if a wedge is *closed*, i.e., there is also edge between u and w, then it corresponds to a triangle. Thus, the number of triangles is proportional to the fraction of closed wedges, which can be approximated by random sampling.

However, to put this idea into practice, one needs to know W, the total number of wedges, as well as a way to sample one uniformly from all the W wedges. The two graph representations do not support such an operation, so an $O(n)$-time and preprocessing step is needed. So strictly speaking, this is not a sublinear-time algorithm. Furthermore, the preprocessing step

will result in an $O(n)$-size array that needs to be kept for the sampling steps.

Note the number of wedges with v as the middle vertex is $\binom{d(v)}{2}$, so $W = \sum_v \binom{d(v)}{2}$, which can be computed in $O(n)$ time. To support uniformly sampling a wedge, we build another array A where $A[v] = \sum_{u=1}^{v} \binom{d(u)}{2}$, i.e., $A[v]$ stores the total number of wedges with u as the middle vertex for all $u \leq v$. To sample a wedge, we generate a wedge index i from 1 to W, and then do a binary search in the array A to locate the vertex v that should serve as the middle vertex of the wedge. After v is decided, we randomly pick two of its neighbors to form the wedge. Then we check if the wedge is closed or not, and scale it up so that it becomes an unbiased estimator of T_3. The detailed algorithm is shown in Algorithm 6.

Algorithm 6: Wedge sampling

```
t ← 0;
sample a wedge (u, v, w) uniformly as described;
if d(u) < d(w) then
    if w ∈ N(u) then t ← 1 ;
else
    if u ∈ N(w) then t ← 1 ;
report t · W/3;
```

The algorithm works in both the edge array model and the adjacency list model. The only difference is that in the edge array model, after the middle vertex v has been decided, the other two vertices can be chosen in $O(1)$ time, whereas in the adjacency list model, $O(d(v))$ time is needed.

4.5.1 Analysis

The probability that $t = 1$ is $3T_3/W$, since a triangle can be found by closing one of 3 wedges, so $t \cdot W/3$ is an unbiased estimator of T_3.

The variance of the estimator t is

$$\begin{aligned}\mathsf{Var}[t] &= \mathsf{E}[t^2] - \mathsf{E}[t]^2 \\ &= \frac{3T_3}{W} - \frac{9T_3^2}{W^2}.\end{aligned}$$

The binary search takes $O(\log n)$ time. To test whether there is an edge between u and w, we scan the neighbor list of the vertex with a smaller degree to check for the other vertex, so the running time is $O(\min(d(u), d(w))$. So, the expected running time of this algorithm is $O(\log n + \frac{1}{W}\sum_{(u,v),(v,w)\in E} \min(d(u), d(w)))$. In the adjacency list model, it becomes $O(\log n + \frac{1}{W}\sum_{(u,v),(v,w)\in E} d(v) + \min(d(u), d(w)))$.

Proposition 6. *The wedge sampling algorithm returns an unbiased estimator of T_3 with variance $\frac{WT_3}{3} - {T_3}^2$. Its running time is $O(\log n + \frac{1}{W}\sum_{(u,v),(v,w)\in E} \min(d(u), d(w)))$ in the edge array model, and $O(\log n + \frac{1}{W}\sum_{(u,v),(v,w)\in E} d(v) + \min(d(u), d(w)))$ in the adjacency list model. This algorithm needs an $O(n)$-time preprocessing step and $O(n)$ working space for the sampling.*

One may wonder if the preprocessing and the additional array A can be avoided, by using non-uniform sampling and compensating the non-uniformity, as we did previously to adapt edge sampling and triangle sampling to the adjacency list model. However, an acute reader may quickly realize that if we do so, wedge sampling will essentially become triangle sampling.

TABLE 1
Summary of datasets. n: the number of vertices, m: the number of edges, T_3: the number of triangles.

Dataset	n	m	T_3
Amazon	3.34×10^5	9.26×10^5	6.67×10^5
Youtube	1.13×10^6	2.99×10^6	3.06×10^6
LiveJournal(LJ)	4.00×10^6	3.47×10^7	1.78×10^8
roadNet-CA	1.97×10^6	2.77×10^6	1.21×10^5
Skitter	1.70×10^6	1.11×10^7	2.88×10^7
USRD	2.39×10^7	2.89×10^7	4.39×10^5
Twitter	3.06×10^7	5.98×10^8	1.87×10^{10}
WebUK	6.23×10^7	9.39×10^8	1.79×10^{11}

5 Experiments

We have implemented all algorithms discussed in Section 4, for both the edge array model and the adjacency list model. This section describes our experimental setup, methodology, and the results. Analysis of the results will be provided in Section 6.

5.1 Setup

We have used a collection of real-world graphs, including social networks, road networks, and autonomous systems graphs, in our experimental study. A summary of these datasets is given in Table 1. The first 5 datasets are obtained from *SNAP* (http://snap.stanford.edu/). The dataset *USRD* and *WebUK* are the same as in [12]. *Twitter* is obtained from [35]. *Amazon* is crawled from Amazon, where nodes represent products and edges indicate commonly co-purchased products. *LiveJournal* is obtained from a free online community (www.livejournal.com), where vertices are members and an edge represents the friendship between two members. *USRD* is the road network of United States and *roadNet-CA* is a network of California, where vertices represent intersections and endpoints, and edges represent the roads connecting these intersections or road endpoints. *WebUK* is a webspam dataset, where vertices are pages and edges are hyperlinks between pages. *Twitter* is an online microblog where vertices are users and edges represents users are followed by others.

All the experiments were performed under CentOS 5.10 (64 bits) on a machine that was running an Intel E5450 3GHz CPU (8 cores) with 16G main memory. All programs were compiled with GNU g++ version 4.9.1 by using flag `-O3`.

We adopt the most standard implementation for the two graph representations. For each graph, the vertex id's are from 1 to n, which are stored as 32-bit integers. For the adjacency list model, the vertex array is implemented as a `vector` using STL, where each entry stores a pointer to a neighbor list. Each neighbor list is implemented as a `list`, which is implemented as a doubly-linked list in STL. For the edge array model, the edge array is a `vector` storing all the neighbor lists in concatenation. The vertex array is a `vector` where each entry stores the index of the first neighbor of the corresponding vertex in the edge array. Note that no pointers are needed in the edge array model. Since the vertex id's are 32-bit integers while pointers are 64-bit long, the size of the edge

array representation is roughly $1/5$ of that of the adjacency list representation for the same graph.

5.2 Methodology

Before running the algorithms, we pre-load the graph from the data file to memory using one of the two representation formats (i.e., loading time is not included in measuring the running time of the algorithms). Note that some large graphs used in our experiments do not fit in main memory; we simply rely on the virtual memory system to handle this automatically.

Recall that all the algorithms run for multiple sampling steps (except the subgraph sampling algorithm), with each step returning an unbiased estimator of T_3. We take the average of these estimators, whose accuracy thus improves as more steps are taken. More precisely, if the variance of the estimator from one sampling step is σ^2, the variance after k steps is σ^2/k. However, since different algorithms have different per-step running time, it will not be a fair comparison to use the same k for all algorithms; even for the same algorithm, the per-step cost also varies from step to step (it is a random variable). Thus, we adopt the following scheme in order to have a fair comparison across different algorithms: Suppose we run algorithm $\mathcal{A}$ on a particular graph, and let $\bar{x}(t)$ be the running average of the sampling steps so far until time t. We calculate the error $\bar{x}(t) - T_3$ at regular time intervals, say $t = 10$ms, 20ms, ... Since one run of the algorithm may have high fluctuation, we repeat the process multiple times, and for each time stamp t, we report the root mean square error (RMSE) of the algorithm across multiple runs, namely,

$$\mathrm{RMSE}(t) = \sqrt{\frac{\sum_{i=1}^{r}(\bar{x}^i(t) - T_3)^2}{r}},$$

where $\bar{x}^i(t)$ is the running average at time t in the i-th run, and there are a total of r runs. In our experiments, we used $r = 100$ runs to get stable results. Furthermore, we report the relative RMSE as a percentage of T_3, so that results across different data sets can be compared.

The reader is reminded that the RMSE is used to evaluate and compare these algorithms. In actual application, we cannot compute RMSE as we do not know T_3. However, since each sampling step of the algorithm returns an unbiased estimator T_3, σ can be estimated as (from standard statistics theory)

$$\tilde{\sigma} = \sqrt{\frac{1}{k-1}\sum_{j=1}^{k}(x_j - \bar{x})^2},$$

where x_j is the estimate returned from the j-th sampling step, and $\bar{x} = \frac{1}{k}\sum_{j=1}^{k} x_j$. When k is sufficiently large, $\tilde{\sigma}/\sqrt{k}$ will be a reasonable estimate of the RMSE.

In order to see the benefits of sublinear-time algorithms, we have also run the exact counting algorithm for each graph. For graphs that fit in memory, we implement the *Compact Forward* algorithm [9]. For larger graphs, we used the external memory algorithm and code from [7]. The running times of these exact counting algorithms are given in Table 2.

5.3 Results

From the experiments, we first observed that the subgraph sampling algorithm is much worse than the other algorithms, to the point that their results cannot be plotted in the same figure. Instead, we indicate the result of the subgraph sampling algorithm in a box in each figure as a (RMSE, running time) pair. For subgraph sampling, we need to decide the sample size k before running the algorithm. In our experiments, we tried various k so that the its running time is on the same order as other algorithms, and report that result for that particular value of k. It turns out with this time constraint, the algorithm can sample no more than 1% of the edges, which results in very poor estimation quality. We note that in the original paper [21], more than 10% of the edges had to be sampled in order to get reasonable estimates, but that would make the algorithm run much slower than the other algorithms. The intuitive reason is that subgraph sampling is too general a technique. It can in fact be used to approximately count the number of *any* subgraph pattern, not just triangles. In doing so, the algorithm "blindly" samples edges. On the other hand, the other algorithms are tailored to finding triangles more intelligently, thus leading to much better accuracy.

The detailed experimental results are plotted in Figure 4 and 5, which show how the RMSE reduces over time for all algorithms on each of the datasets. For the same algorithm, we plot the results under the two different graph representation models in the same figure, so as to see the benefit of using the more compact edge array model. The version for the adjacency list model is shown in dashed lines, while the one for the edge array model in solid lines. Note that the vertex sampling algorithm is the same for both models, so there is only one line for this algorithm[1]. Particularly, we present two more figures: Figure 5(b) and 5(d) to show the details at twisted line part.

TABLE 2
Running times of exact counting algorithms.

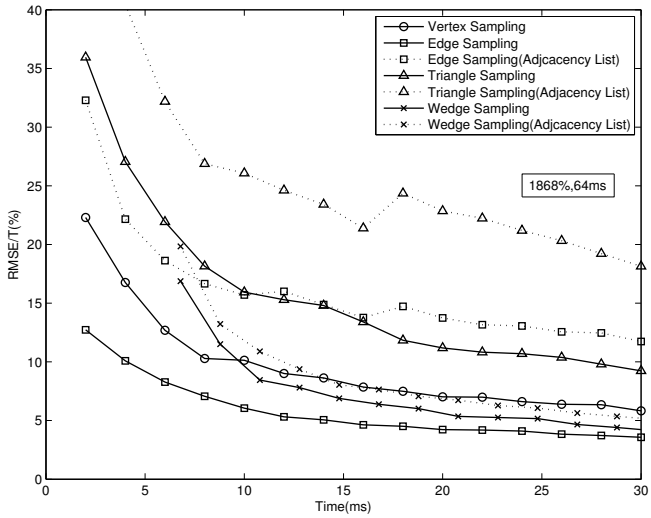
Dataset	Time (s)	Dataset	Time (s)
Amazon	1.1	LiveJournal(LJ)	200
Youtube	42	USRD	37
roadNet-CA	5.5	Twitter	1,950
Skitter	441	WebUK	2,102

TABLE 3
Loading time of datasets

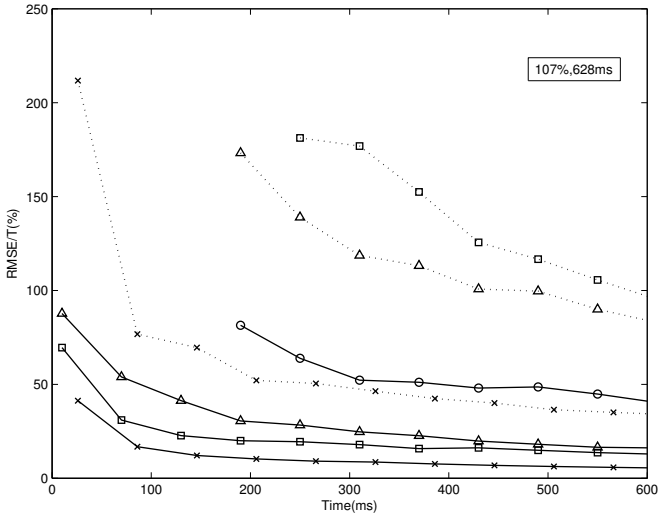
Dataset	Edge array model (ms)	Adjacency list model (ms)
Amazon	331	455
Youtube	505	1,079
LiveJournal(LJ)	3,528	10,764
roadNet-CA	593	1,778
Skitter	1,018	3,551
USRD	11,268	16,383
Twitter	49,173	571,059
WebUK	122,351	1,349,232

1. Strictly speaking, the implementation of the vertex sampling algorithm is still slightly different in the two models, as one uses `list` while the other uses `vector` to store the neighbor lists, and traversing a `vector` is slightly faster than in a `list`. But the difference is very small, hence neglected.

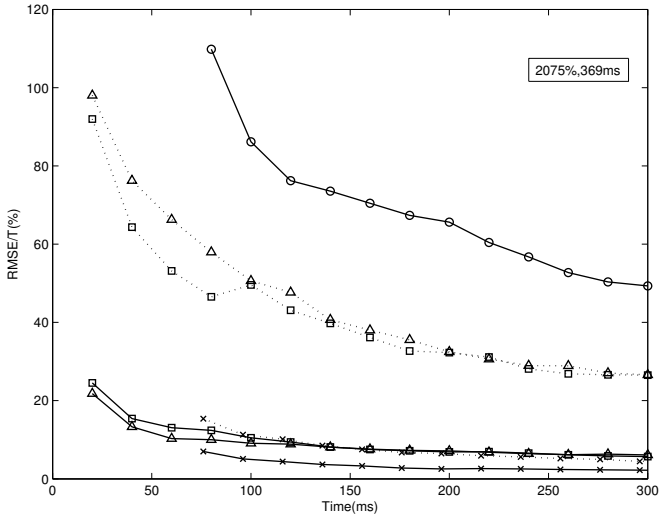
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2016.2556663, IEEE Transactions on Knowledge and Data Engineering



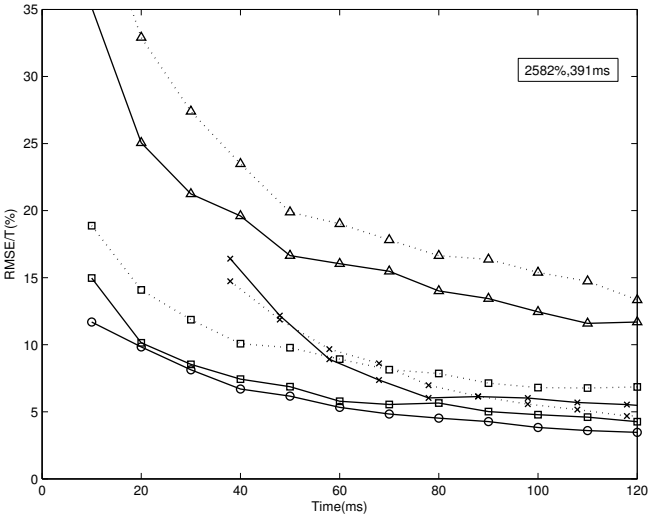
(a) Amazon



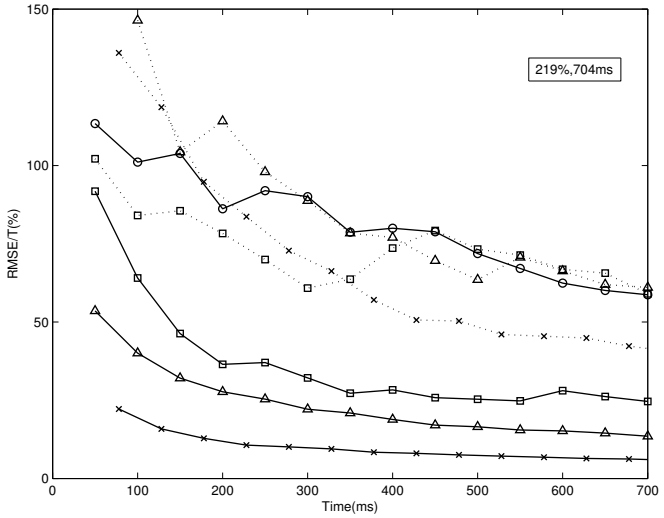
(b) Youtube



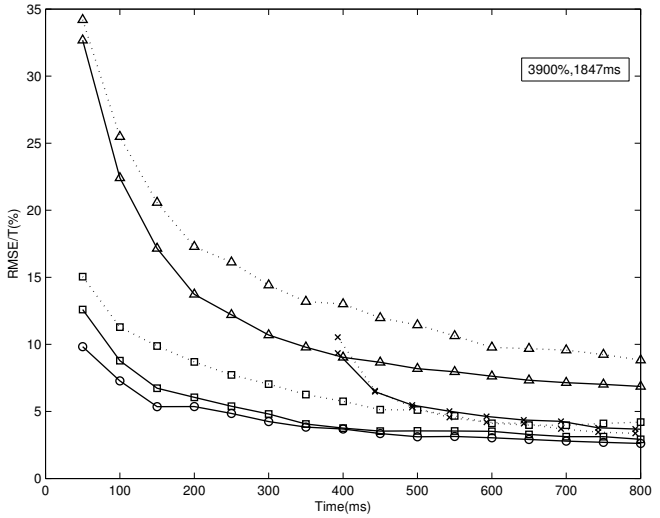
(c) LiveJournal



(d) roadNet-CA



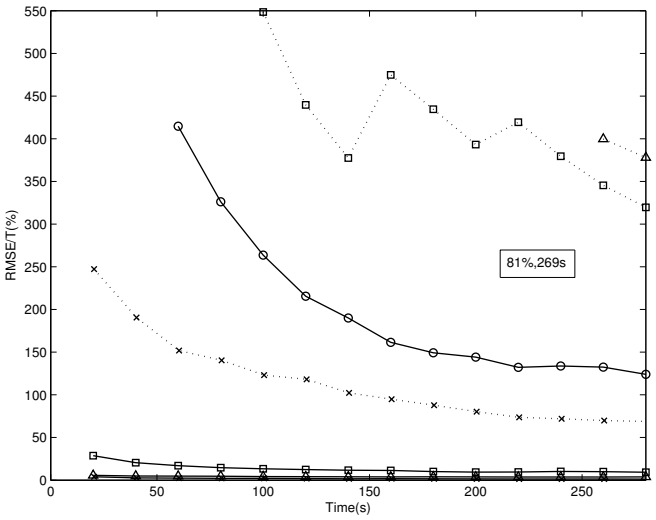
(e) Skitter



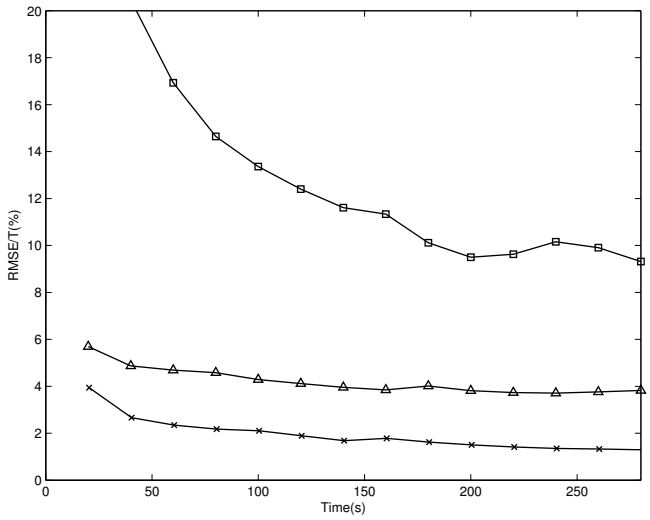
(f) USRD

Fig. 4. Experimental Results (on data sets that fit in main memory)

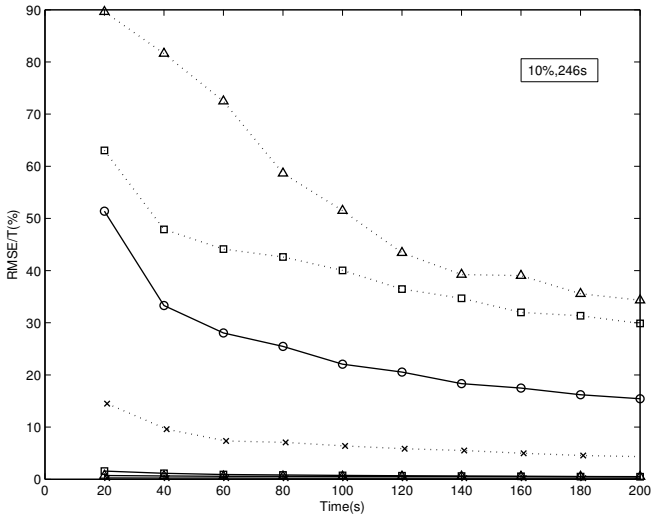
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2016.2556663, IEEE Transactions on Knowledge and Data Engineering



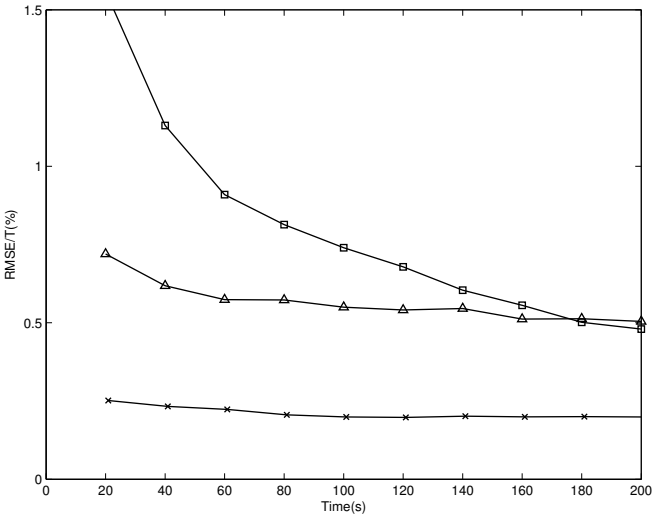
(a) Twitter



(b) Twitter (only showing the competitive algorithms)



(c) WebUK



(d) WebUK (only showing the competitive algorithms)

Fig. 5. Experimental Results (on data sets that do not fit in main memory)

Note that since the wedge sampling algorithm needs an $O(n)$-time preprocessing step, it has a "delayed start" compared with other algorithms. Furthermore, it needs extra $O(n)$ working space to sample the wedges, while the other algorithms only need space to hold the neighbor list of the sampled vertex or edge.

5.4 Experimental observations

From our experimental results on a variety of graphs, we make the following observations.

1) For the same algorithm, the edge array model always offers better performance than the adjacency list model, and the difference can be large on some graphs.
2) Edge sampling and wedge sampling are generally the two best-performing algorithms, with quite stable performance across all data sets. Recall that, however, wedge sampling has a delayed start and needs $O(n)$ working space.
3) Vertex sampling and triangle sampling perform reasonably well on some graphs, but could be a lot worse on other graphs.

6 Analysis

In this section, we try to substantiate the experimental observations made above, through an analytical comparison of these algorithms. The variances and running times of all the algorithms are summarized in Table 4. We have omitted the $-T_3^2$ term from all the variances, which is common to all algorithms, and is insignificant compared with the leading term. The expected running time per sampling step, strictly speaking, should be in "big-Oh" notation. But since all the algorithms perform almost the same type of operations (random sampling followed by hash join between neighbor lists), the hidden constants are very close. Thus we drop the big-Oh and consider these as reasonably good approximations of actual running times.

TABLE 4
Comparison of all algorithms. The $-T_3^2$ term is omitted from all the variances, which is common to all algorithms.

Model	Method	Variance	Expected running time per sampling step
Both	Vertex sampling	$\frac{n}{9}\sum_v \lambda(v)^2$	$\frac{1}{n}\sum_v d(v)^2$
Edge array	Edge sampling	$\frac{m}{9}\sum_e \lambda(e)^2$	$\frac{1}{m}\sum_v d(v)^2$
	Triangle sampling	$\frac{m}{18}\sum_e \lambda(e)(d(u)+d(v))$	$\frac{1}{m}\sum_e \frac{d(u)d(v)}{d(u)+d(v)}$
	Wedge sampling	$\frac{1}{3}WT_3$	$\log n + \frac{1}{W}\sum_{(u,v),(v,w)\in E} \min(d(u),d(w))$
Adjacency list	Edge sampling	$\frac{n}{9}\sum_e \frac{\lambda(e)^2 d(u)d(v)}{d(u)+d(v)}$	$\frac{1}{n}\sum_e \frac{(d(u)+d(v))^2}{d(u)d(v)}$
	Triangle sampling	$\frac{n}{18}\sum_e \lambda(e)d(u)d(v)$	$\frac{1}{n}\sum_e \frac{(d(u)+d(v))^2}{d(u)d(v)}$
	Wedge sampling	$\frac{1}{3}WT_3$	$\log n + \frac{1}{W}\sum_{(u,v),(v,w)\in E} d(v) + \min(d(u),d(w))$

6.1 Variance after a certain amount of time

Table 4 has listed, for each algorithm, the variance of expected running time of a *single* sampling step. These, however, do not yet tell us the actual performance of the algorithms. What if an algorithm has a small variance per sampling step but each sampling step takes more time? Ultimately, what we care is the variance of the final estimator, which is the average of all the sampling steps taken, after a certain amount of time, say t. If each sampling step takes a fixed amount of time T, then we know that there must be $k = t/T$ sampling steps after time t, and the final variance is simply the variance of one sampling step divided by k. However, in our case T is a random variable, which means that the number of sampling steps k is also a random variable, and this introduces some complication.

Let X_i be the estimator yielded in the i-th sampling step. The final estimator after time t is thus $\frac{1}{k}\sum_{i=1}^k X_i$. The following lemma derives its variance.

Proposition 7. *The variance of the estimator after time t is* $\mathsf{Var}\left[\frac{1}{k}\sum_{i=1}^k X_i\right] = \mathsf{Var}[X_1]\mathsf{E}[T]/t$.

Proof. Note that this is a sum of a random number of random variables, so we cannot directly break it up as in standard variance analysis. We will have to do a conditioning on k, and then use the *law of total variance*:

$$\begin{aligned}
&\mathsf{Var}\left[\frac{1}{k}\sum_{i=1}^k X_i\right]\\
=&\mathsf{E}\left[\mathsf{Var}\left[\frac{1}{k}\sum_{i=1}^k X_i \,\middle|\, k\right]\right] + \mathsf{Var}\left[\mathsf{E}\left[\frac{1}{k}\sum_{i=1}^k X_i \,\middle|\, k\right]\right]\\
=&\mathsf{E}[\mathsf{Var}[X_1]/k] + \mathsf{Var}[T_3]\\
=&\mathsf{Var}[X_1]\mathsf{E}[T/t] + 0\\
=&\mathsf{Var}[X_1]\mathsf{E}[T]/t.
\end{aligned}$$

□

Proposition 7 has two implications. First, given the same amount of time t, the performance of the algorithm is determined by $\mathsf{Var}[X_1]\mathsf{E}[T]$. So it is sufficient to use the term $\mathsf{Var}[X_1]\mathsf{E}[T]$ for the comparison of different algorithms. Second, it also gives us a condition of sublinearity. More precisely, suppose we aim at a relative error of ε, i.e., $\mathsf{Var}[X_1]\mathsf{E}[T]/t = (\varepsilon T_3)^2$, then this means that $t = \mathsf{Var}[X_1]\mathsf{E}[T]/(\varepsilon T_3)^2$. By plugging in the $\mathsf{Var}[X_1]$ and $\mathsf{E}[T]$ formulas of various algorithms from Table 4, we can analyze their running times for achieving an ε-approximation for certain classes of graphs. The algorithm can be considered as taking sublinear time if $t = o(n+m)$.

6.2 Sublinearity

Below, we give some examples on how to check the sublinearity of various algorithms for certain classes of graphs.

To start with, consider a complete graph, which has $m = n^2$ edges and $T_3 = n^3$ triangles (we again omit the big-Oh for notational simplicity). Then the edge sampling algorithm (which is the algorithm we advocate the most) in the edge array model has running time

$$\frac{\mathsf{Var}[X_1]\mathsf{E}[T]}{(\varepsilon T_3)^2} = \frac{\sum_e \lambda(e)^2 \sum_v d(v)^2}{\varepsilon^2 n^6} = \frac{n^2\cdot n^2\cdot n\cdot n^2}{\varepsilon^2 n^6} = \frac{n}{\varepsilon^2},$$

which is $o(m+n) = o(n^2)$. So it is a sublinear-time algorithm as long as $\varepsilon > n^{-1/2}$.

On the other, vertex sampling is not a sublinear-time algorithm on a complete graph, since its running time is

$$\frac{\mathsf{Var}[X_1]\mathsf{E}[T]}{(\varepsilon T_3)^2} = \frac{\sum_v \lambda(v)^2 \sum_v d(v)^2}{\varepsilon^2 n^6} = \frac{n\cdot n^4\cdot n\cdot n^2}{\varepsilon^2 n^6} = \frac{n^2}{\varepsilon^2}.$$

Next, consider a triangulation graph of constant degree, which is a common type of sparse graphs with many triangles. In a triangulation, we have $m = n$ and $T_3 = n$. For edge sampling, we have

$$\frac{\mathsf{Var}[X_1]\mathsf{E}[T]}{(\varepsilon T_3)^2} = \frac{\sum_e \lambda(e)^2 \sum_v d(v)^2}{\varepsilon^2 n^2} = \frac{n\cdot 1^2\cdot n\cdot 1^2}{\varepsilon^2 n^2} = \frac{1}{\varepsilon^2}.$$

This is a very nice result as it indicates that its running time is *independent* of the graph size, and is only determined by the desired error level.

The vertex sampling on a planar triangulation graph can do as well, since we similarly have

$$\frac{\mathsf{Var}[X_1]\mathsf{E}[T]}{(\varepsilon T_3)^2} = \frac{\sum_v \lambda(v)^2 \sum_v d(v)^2}{\varepsilon^2 n^2} = \frac{n\cdot 1^2\cdot n\cdot 1^2}{\varepsilon^2 n^2} = \frac{1}{\varepsilon^2}.$$

Actually, later we will prove a strong result that no matter what the graph is, edge sampling will always do at least as well as vertex sampling (c.f. Proposition 8).

In general, however, it is difficult to check sublinearity for an arbitrary graph, and the analyses above rely on some fairly strong properties of the class of graphs under investigation. On the other hand, we have computed the term $\mathsf{Var}[X_1]\mathsf{E}[T]/(\varepsilon T_3)^2$ for the 8 real graphs used in our experimental study (see Table 5) for the edge sampling algorithm, which can serve as its empirical evidence of sublinearity for typical real-world graphs.

TABLE 5
Comparison of $\mathsf{Var}[X_1]\mathsf{E}[T]/(\varepsilon T_3)^2$ ($\varepsilon = 0.5$)

Model	Method	Amazon	Youtube	LJ	roadNet-CA	Skitter	USRD	twitter	WebUK
Both	Vertex sampling	4.12×10^3	4.94×10^7	7.05×10^6	3.91×10^3	2.45×10^8	2.12×10^3	1.05×10^9	6.58×10^7
Edge array	Edge sampling	1.93×10^2	7.21×10^4	9.6×10^3	4.14×10^2	7.08×10^5	5.01×10^2	1.18×10^5	1.15×10^4
	Triangle sampling	1.63×10^2	7.53×10^4	5.28×10^3	6.14×10^2	8.0×10^4	5.09×10^2	2.66×10^5	6.36×10^3
	Wedge sampling	5.99×10^2	2.78×10^4	3.91×10^3	2.83×10^3	4.70×10^4	5.64×10^3	1.49×10^5	9.10×10^3
Adjacency list	Edge sampling	1.08×10^3	4.64×10^7	1.68×10^5	9.88×10^2	7.24×10^8	1.24×10^3	4.75×10^9	1.29×10^6
	Triangle sampling	3.30×10^3	5.27×10^8	3.40×10^5	3.34×10^3	1.44×10^9	3.92×10^3	2.35×10^{10}	1.50×10^6
	Wedge sampling	1.49×10^3	7.55×10^6	5.07×10^4	3.39×10^3	1.35×10^7	6.13×10^3	2.86×10^5	1.40×10^4

6.3 Edge array vs. adjacency list

Our first experimental observation was that, for the same algorithm, the edge array model always offers better performance than the adjacency list model. Practically, this is because the edge array model is more compact. It uses arrays to store the neighbor lists, which allows more cache-efficient traversal of neighbors. On the other hand, the adjacency list model uses linked lists to store the neighbor lists, which is less cache-efficient as it involves pointer-jumping during traversal.

Below we also provide theoretical justification on why the edge array model is better. As argued above, we can compare $\mathsf{Var}[X_1]\mathsf{E}[T]$ for the same algorithm under the two models. The comparison for the wedge sampling algorithm is straightforward: $\mathsf{Var}[X_1]$ is the same under the two models, while $\mathsf{E}[T]$ is strictly larger in the adjacency list model.

The comparison for the edge sampling algorithm is more subtle. In fact, there is no strict winner in all cases. Consider the two extreme examples in Figure 6 and 7.

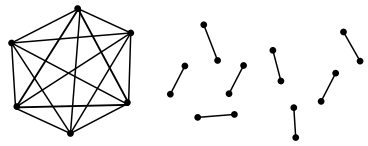


Fig. 6. A graph that consists of a complete graph K_n and n^3 single edges.

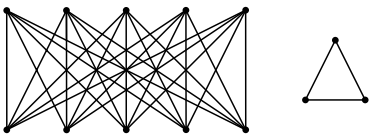


Fig. 7. A graph that consists of a complete bipartite graph $K_{n,n}$ and one triangle.

From the analytical results in Table 4, we know that $\mathsf{Var}[X_1]\mathsf{E}[T] = \sum_e \lambda(e)^2 \sum_v d(v)^2$ in the edge array model, and $\mathsf{Var}[X_1]\mathsf{E}[T] = \sum_e \frac{\lambda(e)^2 d(u)d(v)}{d(u)+d(v)} \sum_e \frac{(d(u)+d(v))^2}{d(u)d(v)}$ in the adjacency list model (ignoring the common coefficient $\frac{1}{9}$). For the graph in Figure 6, $\mathsf{Var}[X_1]\mathsf{E}[T] = \Theta(n^7)$ in the edge array model while it is $\Theta(n^8)$ in the adjacency list mode. However, for the graph in Figure 7, $\mathsf{Var}[X_1]\mathsf{E}[T] = \Theta(n^3)$ in the edge array model while it is $\Theta(n^2)$ in the adjacency list model.

The above two extreme examples imply that it is not possible to prove that one model is always better than the other. Our experimental results, on the other hand, seem to have suggested that the edge array model is better than the adjacency list model, meaning that real graphs are more similar to Figure 6 than to Figure 7. Indeed, on the class of graphs we have experimented with (social networks, relationship between products), vertices tend to form small tightly connected clusters, while bipartite graphs are rare. This in turn means that more triangles tend to occur around high-degree vertices (bipartite graphs exactly lack this property). If we assume that for each edge $e = (u, v)$, $\lambda(e)$ is proportional to $d(u)$ and $d(v)$, then we can prove that the edge array model is indeed better. Suppose $d(u)/\lambda(e) \approx d(v)/\lambda(e) \approx c$. Then for the edge array model,

$$\begin{aligned}\mathsf{Var}[X_1]\mathsf{E}[T] &= \sum_e \lambda(e)^2 \sum_e (d(u)+d(v)) \\ &\approx c\sum_e \lambda(e)^2 \sum_e \lambda(e).\end{aligned}$$

For the adjacency list model, we have

$$\mathsf{Var}[X_1]\mathsf{E}[T] \approx \sum_e c\lambda(e)^3 \sum_e 1 = cm\sum_e \lambda(e)^3.$$

By Chebyshev's sum inequality, we have

$$\sum_e \lambda(e)^2 \sum_e \lambda(e) \le m\sum_e \lambda(e)^3,$$

so $\mathsf{Var}[X_1]\mathsf{E}[T]$ is always smaller in the edge array model.

The triangle sampling algorithm can be similarly analyzed. Again, for the graph in Figure 6, the edge array model is better than the adjacency list model by an order of $\Theta(n)$, but is worse by an order of $\Theta(n)$ for the graph in Figure 7. Still, with the assumption that $\lambda(e)$ is proportional to $d(u)$ and $d(v)$, we can show that the edge array model is better.

Therefore, we can conclude that for all the algorithms, the edge array model offers better performance than the adjacency list model, for most real-world graphs where more triangles tend to occur around high-degree vertices. The edge array model is also a more compact and more cache-efficient graph representation. Thus, we focus on the edge array model for the rest of the analysis.

6.4 Edge sampling

Our experimental results suggest that edge sampling is always one of the best performing algorithms across all the data sets, with no need for preprocessing and very little working space. In this section, we will justify this claim analytically.

6.4.1 Comparison with vertex sampling

From Table 4, the comparison between edge sampling and vertex sampling in terms of $\mathsf{Var}[X_1]\mathsf{E}[T]$ boils down to comparing $\sum_e \lambda(e)^2$ and $\sum_v \lambda(v)^2$. We have the following fairly strong result, which holds for all graphs.

Proposition 8. *For any graph G, $\sum_e \lambda(e)^2 \leq \sum_v \lambda(v)^2$.*

Proof. First, observe that $\lambda(v) = \frac{1}{2}\sum_{u\in N(v)} \lambda(u,v)$. So we have

$$\begin{aligned}\sum_v \lambda(v)^2 =& \sum_v \left(\frac{1}{2}\sum_{u\in N(v)} \lambda(u,v)\right)^2 \\ =& \frac{1}{4}\sum_v \Big(\sum_{u\in N(v)} \lambda(u,v)^2 \\ &+ \sum_{u\neq w\in N(v)} \lambda(u,v)\lambda(v,w)\Big).\end{aligned}$$

Obviously, $\sum_{v\in V}\sum_{u\in N(v)} \lambda(u,v)^2 = 2\sum_{e\in E}\lambda(e)^2$. The second term $\sum_v \sum_{u\neq w\in N(v)} \lambda(u,v)\lambda(v,w)$ can be rewritten as

$$\sum_{e=(u,v)} \lambda(e)\Big(\sum_{p\in N(u)-\{v\}} \lambda(p,u) + \sum_{q\in N(v)-\{u\}} \lambda(q,v)\Big).$$

Please see Figure 8. Any triangle counted in $\lambda(e=(u,v))$ is also counted in some $\lambda(p,u)$ for some $p \in N(u)-\{v\}$, so $\sum_{p\in N(u)-\{v\}} \lambda(p,u) \geq \lambda(e)$. Similarly, $\sum_{q\in N(v)-\{u\}} \lambda(q,v) \geq \lambda(e)$. Thus, the second term $\sum_v \sum_{u\neq w\in N(v)} \lambda(u,v)\lambda(v,w)\} \geq 2\sum_e \lambda^2(e)$. The proof thus completes after combining the two parts. □

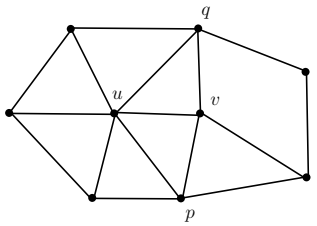


Fig. 8. Vertex Sampling and Triangle Sampling

Therefore, we conclude that edge sampling is better than vertex sampling on any graph.

6.4.2 Comparison with triangle sampling

Unfortunately, it is not possible to show that edge sampling is always better than triangle sampling. In fact, on the extreme example in Figure 9, $\mathsf{Var}[X_1]\mathsf{E}[T] = \Theta(n^4)$ for edge sampling but $\mathsf{Var}[X_1]\mathsf{E}[T] = \Theta(n^3)$ for triangle sampling. This is actually quite intuitive. This graph has one crucial edge that is shared by all the triangles. Edge sampling has to sample that edge in order to find any triangles, which happens with probability $\Theta(1/n)$. On the other hand, the triangle sampling algorithm can always find triangles no matter what vertex is sampled.

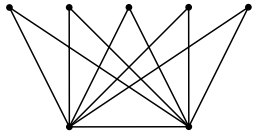


Fig. 9. A graph that consists of n triangles sharing a single edge.

However, if we assume that the degrees of neighboring vertices do not differ too much (within a constant factor), we can indeed show that edge sampling is always better. Under this assumption, the $\mathsf{E}[T]$ part of both algorithms are roughly the same, both being $\sum_v d(v)^2 = \sum_e d(u) + d(v)$. For the $\mathsf{Var}[X_1]$ part, compared with edge sampling, triangle sampling replaces one $\lambda(e)$ term with $d(u) + d(v)$, which is always greater than $\lambda(e)$. This actually partially explains why triangle sampling performs much worse on the road network graphs (roadNet-CA, USRD), which have relatively few triangles, so $\lambda(e)$ is much smaller than $d(u)+d(v)$. On the other hand, when neighboring triangles have very different degrees, such as the extreme example in Figure 9, triangle sampling can perform better. The real graph Skitter also has similar properties, so triangle sampling performs relatively better on it than on other graphs.

6.5 Wedge sampling

The performance of wedge sampling is characterized by parameters different from those for other algorithms, and the formulas for $\mathsf{Var}[X_1]$ and $\mathsf{E}[T]$ are also quite different from others. Making things even worse, as wedge sampling requires an $O(n)$-time preprocessing step, the $\mathsf{Var}[X_1]\mathsf{E}[T]$ argument used earlier cannot be applied anymore. Thus, unfortunately we cannot have a good analytical comparison between this algorithm and the others. Experimentally, its performance appears to be similar to that of edge sampling, albeit with a delayed start due to the preprocessing. Furthermore, it requires $O(n)$ working space, while other algorithms require very small working space.

6.6 A recent theoretical result

Very recently, Eden et al. [8] gave an algorithm for approximating the number of triangles in $O(\mathrm{poly}(\varepsilon^{-1}\log n)(n/T_3^{1/3} + \min\{m, m^{3/2}/T_3\}))$ time, which is sublinear when $T_3 \gg \sqrt{m}$. Theoretically speaking, this result is much more elegant as it only depends on T_3 (other than the input size), whereas the bounds of our algorithms are a lot messier. Still, this result does not subsume ours. For example, on triangulation graphs, we showed previously that edge sampling has running time $O(1/\varepsilon^2)$, while the Eden et al. bound is $O(\mathrm{poly}(\varepsilon^{-1}\log n)n^{2/3})$.

We have also examined the practicality of the Eden et al. algorithm. After carefully analyzing their algorithm, we have derived the poly factor in the time complexity, and the full running time is $O(\varepsilon^{-4}\log^3 n \log\log n(n/T_3^{1/3} + \min\{m, m^{3/2}/T_3\}))$. On a typical graph, e.g., the Twitter data set, which has (roughly) $n = 10^7, m = 10^9, T_3 = 10^{10}$, with an $\varepsilon = 10\%$ error, this running time is on the order of 10^{12}, which is actually much larger than the input size.

To check whether this large running time might have been due to the slack in the analysis, we have also implemented the algorithm (the "simpler" version as described in [36]). Note that, besides listing all neighbors of a given vertex, this algorithm also requires to check the existence of an edge (u,v) for a given u and v, which is not supported directly by the edge array or the edge list representation. So we built hash tables on the neighbor list of every vertex (the time to build the hash tables is not included in the reported times). This roughly doubles the storage cost of the graph. We set $\varepsilon = 10\%$ and ran the algorithm on the data sets, and the running times are reported in Table 6. We see that it is even slower than the

exact counting algorithms, as reported in Table 2. Therefore, we conclude that this algorithm is of theoretical interests only.

TABLE 6
Running times of the Eden et al. algorithm.

Dataset	Time (s)	Dataset	Time (s)
Amazon	39	LiveJournal(LJ)	74
Youtube	140	USRD	3,448
roadNet-CA	183	Twitter	4,057
Skitter	55	WebUK	>259,200

7 Conclusions

In this paper, we have provided a detailed experimental and analytical comparison of different approaches to the approximate triangle counting problem by random sampling. Our results suggest that edge sampling is a good candidate for a variety of graphs, with both experimental and analytical evidence. Wedge sampling is also quite competitive, if a delayed start is tolerable, although a good analytical understanding of its performance remains elusive.

A very interesting problem to be further investigated is to see if there are better techniques on sampling from disk-resident graph data. In this paper we have simply relied on the virtual memory system to handle graphs that do not fit in memory. This essentially means that upon a page fault, the algorithm will go to disk and fetch the page containing the sampled vertex or edge, but will not utilize the rest of the data on that page (unless some other data is luckily sampled again). Ideally, sampling from disk-resident data should be block-based. However, the problem is that data from the same block may not be independent. In the worst case, if the data within one block are highly correlated, then using all data from the sampled block is the same as using just one record from it. For a linear array, there has been work on how to better exploit block-based sampling [37], [38]. However, as graph data is much more complicated than a linear array, the problem of block-based sampling on graphs is more challenging but certainly very interesting.

Acknowledgments

This work is supported by HKRGC under grants GRF-621413, GRF-16211614, GRF-16200415, and by Huawei Research Fund.

References

[1] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-worldnetworks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.
[2] T. Opsahl and P. Panzarasa, "Clustering in weighted networks," *Social networks*, vol. 31, no. 2, pp. 155–163, 2009.
[3] J.-P. Eckmann and E. Moses, "Curvature of co-links uncovers hidden thematic layers in the world wide web," *Proceedings of the national academy of sciences*, vol. 99, no. 9, pp. 5825–5829, 2002.
[4] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *SIGKDD*, 2008, pp. 16–24.
[5] J. W. Berry, B. Hendrickson, R. A. LaViolette, and C. A. Phillips, "Tolerating the community detection resolution limit with edge weighting," *Physical Review E*, vol. 83, no. 5, p. 056119, 2011.
[6] A. Itai and M. Rodeh, "Finding a minimum circuit in a graph," in *STOC*, 1977, pp. 1–10.
[7] X. Hu, Y. Tao, and C.-W. Chung, "Massive graph triangulation," in *SIGMOD*, 2013, pp. 325–336.
[8] T. Eden, A. Levi, D. Ron, and C. Seshadhri, "Approximately counting triangles in sublinear time," in *FOCS*, 2015.
[9] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theor. Comput. Sci.*, vol. 407, no. 1-3, pp. 458–473, 2008.
[10] T. Schank and D. Wagner, "Finding, counting and listing all triangles in large graphs, an experimental study," in *Experimental and Efficient Algorithms*. Springer, 2005, pp. 606–609.
[11] X. Hu, M. Qiao, and Y. Tao, "Join dependency testing, loomis-whitney join, and triangle enumeration," in *PODS*, 2015.
[12] S. Chu and J. Cheng, "Triangle listing in massive networks," *ACM Transactions on Knowledge Discovery from Data*, vol. 6, no. 4, p. 17, 2012.
[13] R. Dementiev, "Algorithm engineering for large data sets hardware, software, algorithms," *PhD thesis, Saarland University*, 2006.
[14] B. Menegola, "An external memory algorithm for listing triangles," *Technical report, Universidade Federal do Rio Grande do Sul*, 2010.
[15] R. Pagh and F. Silvestri, "The input/output complexity of triangle enumeration," in *PODS*, 2014, pp. 224–233.
[16] R. Pagh and C. E. Tsourakakis, "Colorful triangle counting and a mapreduce implementation," *Information Processing Letters*, vol. 112, no. 7, pp. 277–281, 2012.
[17] H.-M. Park, F. Silvestri, U. Kang, and R. Pagh, "Mapreduce triangle enumeration with guarantees," in *CIKM*, 2014, pp. 1739–1748.
[18] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *WWW*, 2011, pp. 607–614.
[19] J.-H. Yoon and S.-R. Kim, "Improved sampling for triangle counting with mapreduce," in *Convergence and Hybrid Information Technology*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, vol. 6935, pp. 685–689.
[20] A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman, "Upper and lower bounds on the cost of a map-reduce computation," *Proceedings of the VLDB Endowment*, vol. 6, no. 4, pp. 277–288, 2013.
[21] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "Doulion: counting triangles in massive graphs with a coin," in *SIGKDD*, 2009, pp. 837–846.
[22] C. Seshadhri, A. Pinar, and T. G. Kolda, "Triadic measures on graphs: the power of wedge sampling," in *SDM*, 2013.
[23] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis, "Efficient triangle counting in large graphs via degree-based vertex partitioning," *Internet Mathematics*, vol. 8, no. 1-2, pp. 161–185, 2012.
[24] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *SODA*, 2002, pp. 623–632.
[25] H. Jowhari and M. Ghodsi, "New streaming algorithms for counting triangles in graphs," in *Computing and Combinatorics*. Springer, 2005, pp. 710–716.
[26] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, "Counting triangles in data streams," in *PODS*, 2006, pp. 253–262.
[27] D. M. Kane, K. Mehlhorn, T. Sauerwald, and H. Sun, "Counting arbitrary subgraphs in data streams," in *Automata, Languages, and Programming*. Springer, 2012, pp. 598–609.
[28] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Counting and sampling triangles from a graph stream," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1870–1881, 2013.
[29] M. Jha, C. Seshadhri, and A. Pinar, "A space efficient streaming algorithm for triangle counting using the birthday paradox," in *SIGKDD*, 2013, pp. 589–597.
[30] G. Cormode and H. Jowhari, "A second look at counting triangles in graph streams," *Theoretical Computer Science*, vol. 552, pp. 44–51, 2014.
[31] M. Gonen, D. Ron, and Y. Shavitt, "Counting stars and other small subgraphs in sublinear-time," *SIAM Journal on Discrete Mathematics*, vol. 25, no. 3, pp. 1365–1411, 2011.
[32] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010, pp. 135–146.
[33] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *CoRR*, vol. abs/1006.4990, 2010.
[34] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," in *STOC*, 1998, pp. 327–336.
[35] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW*, 2010, pp. 591–600.
[36] C. Seshadhri, "A simpler sublinear algorithm for approximating the triangle count," in *CoRR*, 2015.

[37] S. Chaudhuri, G. Das, and U. Srivastava, "Effective use of block-level sampling in statistics estimation," in *SIGMOD*, 2004.
[38] A. Andoni, P. Indyk, K. Onak, and R. Rubinfeld, "External sampling," in *ICALP*, 2009, pp. 83–94.

Bin Wu is currently a PhD student in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. He obtained his Bachelor's degree from Fudan University in 2012.

Ke Yi is now an Associate Professor in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. He obtained his B.E. from Tsinghua University and Ph.D. from Duke University, in 2001 and 2006 respectively, both in computer science. Before joining HKUST, he was a researcher in the database department at AT&T Labs. His research focus is on big data algorithms and their applications in database systems.

Zhenguo Li is currently a researcher in Huawei Noahs Ark Lab at Hong Kong. He received the B.S. and M.S. degrees from the Department of Mathematics at Peking University, in 2002 and 2005, respectively, and the Ph.D. degree from the Department of Information Engineering at the Chinese University of Hong Kong, in 2008. He was an associate research scientist in the Department of Electrical Engineering at Columbia University. His research interests include machine learning and artificial intelligence.