Towards Optimal Dynamic Indexes for Approximate (and Exact) Triangle Counting

Shangqi Lu
Chinese University of Hong Kong
sqlu@cse.cuhk.edu.hk

Yufei Tao
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

Abstract

In ICDT'19, Kara, Ngo, Nikolic, Olteanu, and Zhang gave a structure which maintains the number T of triangles in an undirected graph $G = (V, E)$ along with the edge insertions/deletions in G. Using $O(m)$ space ($m = |E|$), their structure supports an update in $O(\sqrt{m}\log m)$ amortized time which is optimal (up to polylog factors) subject to the OMv-conjecture (Henzinger, Krinninger, Nanongkai, and Saranurak, STOC'15). Aiming to improve the update efficiency, we study:

- *the optimal tradeoff between update time and approximation quality.* We require a structure to provide the (ϵ, Γ)*-guarantee*: when queried, it should return an estimate t of T that has relative error at most ϵ if $T \geq \Gamma$, or an absolute error at most $\epsilon \cdot \Gamma$, otherwise. We prove that, under any $\epsilon \leq 0.49$ and subject to the OMv-conjecture, no structure can guarantee $O(m^{0.5-\delta}/\Gamma)$ expected amortized update time and $O(m^{2/3-\delta})$ query time simultaneously for any constant $\delta > 0$; this is true for $\Gamma = m^c$ of any constant c in $[0, 1/2)$. We match the lower bound with a structure that ensures $\tilde{O}((1/\epsilon)^3 \cdot \sqrt{m}/\Gamma)$ amortized update time with high probability, and $O(1)$ query time.
- *(for exact counting) how to achieve arboricity-sensitive update time.* For any $1 \leq \Gamma \leq \sqrt{m}$, we describe a structure of $O(\min\{\alpha m + m\log m, (m/\Gamma)^2\})$ space that maintains T precisely, and supports an update in $\tilde{O}(\min\{\alpha + \Gamma, \sqrt{m}\})$ amortized time, where α is the largest arboricity of G in history (and does not need to be known). Our structure reconstructs the aforementioned ICDT'19 result up to polylog factors by setting $\Gamma = \sqrt{m}$, but achieves $\tilde{O}(m^{0.5-\delta})$ update time as long as $\alpha = O(m^{0.5-\delta})$.

2012 ACM Subject Classification Theory of computation → Database query processing and optimization (theory)

Keywords and phrases Triangle Counting, Data Structures, Lower Bounds, Graph Algorithms

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

In the *dynamic approximate triangle counting* (DATC) problem, we want to maintain a data structure on an undirected graph $G = (V, E)$ to support

- **update**(e): either adds a new edge e or removes an existing edge e;
- **query**: returns an estimate t of the number T of *triangles* (i.e., 3-cliques) in G. Specifically, setting $m = |E|$, we require that the estimate t should satisfy an $(\epsilon, \Gamma(m))$*-guarantee*:

$$|t - T| \quad \leq \begin{cases} \epsilon \cdot T & \text{if } T \geq \Gamma(m) \\ \epsilon \cdot \Gamma(m) & \text{otherwise} \end{cases} \tag{1}$$

where ϵ is a parameter of the structure satisfying $0 < \epsilon \leq 1$, and $\Gamma(m)$ a non-descending function of m satisfying

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- $\Gamma(m) \geq 1$
- $\Gamma(c \cdot m) = O(\Gamma(m))$ for any constant $c > 1$.

The query is allowed to fail with probability at most $1/m^2$.

Unless there is a need to emphasize on the parameter m, we will write function $\Gamma(m)$ simply as Γ. The (ϵ, Γ)-guarantee, phrased differently, requires that the estimate t should have a relative error at most ϵ or an absolute error at most $\epsilon \cdot \Gamma$.

The *dynamic exact triangle counting* (DETC) problem is defined analogously except that the value t returned by a query should always be equal to T.

Notations and math conventions. Throughout the paper, $\mathbb{N}$ is the set of integers, $[x]$ denotes the set $\{1, 2, ..., x\}$ for an integer $x \geq 1$, $\tilde{O}(.)$ suppresses a $\text{polylog}\, m$ factor, $\{u, v\}$ represents an undirected edge between vertices u and v, while a directed edge from u to v is represented as (u, v). An event occurs *with high probability* (w.h.p.) if its probability is at least $1 - 1/m^2$.

1.1 Motivation

Triangle counting is equivalent to computing the output size of the conjunctive query

$$ans(a, b, c) \quad = \quad R_1(a, b), R_2(a, c), R_3(b, c). \tag{2}$$

DETC can be easily reduced to the above query by duplicating E three times. Conversely, query (2) can be reduced to DETC as follows. Suppose that relations R_1, R_2, R_3 have schemes $\{\texttt{A}, \texttt{B}\}$, $\{\texttt{A}, \texttt{C}\}$, and $\{\texttt{B}, \texttt{C}\}$, respectively, where attributes A, B, and C have disjoint domains. Create a graph $G = (V, E)$ such that (i) V contains a vertex for every distinct value of A, B, C, and (ii) E has an edge $\{u, v\}$ for every tuple (u, v) of R_1, R_2, R_3. It is easy to verify that each tuple (a, b, c) in the query result corresponds to a unique triangle in G, and vice versa. Inserting/deleting a tuple is translated to an edge update in G.

Our initial motivation stemmed from two recent results on DETC. Subject to the *OMv conjecture* (Section 1.2), Henzinger, Krinninger, Nanongkai, and Saranurak showed [20] (long version [21]) that no structure with $O(m^{0.5-\delta})$ amortized update time can guarantee $O(m^{1-\delta})$ query time, for any constant $\delta > 0$. Kara, Ngo, Nikolic, Olteanu, and Zhang [27] matched this lower bound with a linear-space structure of $O(\sqrt{m} \log m)$ amortized update time[1] and $O(1)$ query time.

$O(\sqrt{m} \log m)$ update time is rather expensive for practical applications. We therefore ask:

> **Question 1:** How much loss of accuracy is necessary, if we want to (significantly) reduce the update cost of [27]?
> **Question 2:** If we insist on exact counting, how to derive an update bound using certain intrinsic parameters of G which can be $o(\sqrt{m})$ for many practical inputs?

1.2 Related Work

Upper Bounds. Kopelowitz et al. [28] studied the following *dynamic set intersection size* problem. Define $\mathcal{C}$ as a collection of non-empty sets $S_1, S_2, ..., S_\ell$ for some $\ell \geq 1$ (the domain of the elements therein is unimportant). Set $m = \sum_{S \in \mathcal{C}} |S|$. Given distinct $i, j \in [\ell]$, a *query* reports the number of elements in $S_i \cap S_j$. We want to maintain a structure to support not only queries and also updates (element insertions/deletions) in the sets of $\mathcal{C}$. The structure of [28] uses $O(m)$ space, performs an

[1] In [27], the amortized update complexity was stated as $O(\sqrt{m})$, assuming that dictionary search on a set of elements can be performed in constant time by a structure that can be updated also in constant time. Removing the assumption with hashing would degrade the update guarantee into an expected bound; doing so with a binary search tree would introduce a logarithmic factor.

update in $\tilde{O}(\sqrt{m})$ time, and answers a query in $\tilde{O}(\sqrt{m})$ time.[2] This structure can be deployed to perform DETC with the same guarantees as [27], up to polylog factors.

Eppstein and Spiro [17] described a DETC structure that supports a query in $O(1)$ time, and an update in $O(h \log m)$ time, where h is the *h-index* of G at the time of the update.[3] The update cost compares favorably with the structure of [27] (Section 1.1) because h is always $O(\sqrt{m})$ but can be far less than $\sqrt{m}$. However, the structure of [17] consumes $O(mh)$ space, while that of [27] needs only $O(m)$ space.

The DETC problem — equivalently, conjunctive query (2) — is a special form of the first-order queries studied by Berkholz et al. [8]. When applied to DETC, their structure performs an update in $\tilde{O}(1)$ time and a query in constant time, when the maximum degree d of the vertices is a constant. In general, however, the update time of [8] is $2^{d^{O(1)}}$ which is much higher than $\sqrt{m}$ even for moderate d. Note that the objective of [8] is to achieve results of this form over a broad class of queries on *sparse* databases (rather than just DETC).

In the *static* scenario where no updates are allowed, the fastest algorithm for exact triangle counting is still the classic $O(m^{2\omega/(\omega+1)})$-time algorithm of Alon, Yuster, and Zwick [1], where $\omega < 2.373$ is the exponent of matrix multiplication. Chiba and Nishizeki [13] described an algorithm of time $O(\alpha m)$ where α is the *arboricity* of G, which is the smallest number of edge-disjoint forests that cover all the edges in G; in general, α is between 1 and $\lceil \sqrt{m} \rceil$. For approximate counting up to relative error ϵ, Eden, Levi, Ron, and Seshadhri [16] gave an algorithm of $\tilde{O}((1/\epsilon)^2 \cdot m^{1.5}/T)$ time. This result can be generalized to counting arbitrary subgraphs; see the work of Assadi, Kapralov, and Khanna [2] and of Chen and Yi [12].

There is a line of research on approximate triangle counting with a *stream algorithm* that makes one or constant passes over E (see [3–5, 9, 11, 15, 18, 23–25, 31, 34, 35, 37] and the references therein). The main purpose there is to minimize the amount of space used. *One-pass* algorithms on *arbitrarily-ordered* streams (i.e., edges arriving in any order) can be used to deal with DATC when only insertions are present. However, in that scenario, Braverman, Ostrovsky, and Vilenchik [9] showed that $\Omega(m)$ space is compulsory even to distinguish between $T = 0$ and $T = \Omega(|V|)$. This implies the necessity of retaining E entirely in the worst case. Our DATC problem complements [9] by asking: as E must be stored anyway, how to organize it properly to permit fast updates?

There have been works on approximate triangle counting on a *dynamic* stream (arbitrary edges insertions and deletions). Bulteau, Froese, Kutzkov, and Pagh [10] developed a structure of $\tilde{O}((1/\epsilon)^2 \cdot \sqrt{m} \cdot P_2/T)$ space that has constant query time but $\tilde{O}((1/\epsilon)^2 \cdot P_2/T)$ update time, where P_2 is the number of 2-paths in G. Another structure due to Manjunath, Mehlhorn, Panagiotou, and Sun [30] uses $\tilde{O}(\text{poly}(1/\epsilon) \cdot m^3/T^2)$ space, and achieves constant query time and $\tilde{O}(\text{poly}(1/\epsilon) \cdot m^3/T^2)$ update time (see also [26]). These structures are applicable to DATC, but their update time is quite large compared to our results (Section 1.3). It should be noted, however, that the focus of [10, 26, 30] is to understand when the space can be made $o(m)$, rather than the update-query tradeoff.

A natural attempt to perform DATC on $G = (V, E)$ is to take a random subset $E' \subseteq E$, build an *exact* counting structure to monitor the number T' of triangles in $G' = (V, E')$, and then scale T' up appropriately to estimate the number of triangles in G. To our knowledge, the most promising approach in this direction is the *colorful triangle sampling* technique by Pagh and Tsourakakis [32], originally proposed for parallel computation. In our contexts, the technique is applicable if Γ is sufficiently large. This can be best illustrated by fixing ϵ to a constant; when $\Gamma \geq c|V| \log_2 |V|$ for

[2] Precisely speaking, Kopelowitz et al. [28] considered a different type of queries, which return *whether* $S_i \cap S_j$ is empty (as opposed to $|S_i \cap S_j|$). However, their structure can be easily adapted to achieve the stated guarantees on the dynamic set intersection size problem.

[3] The h-index is the maximum integer x such that G has x vertices of degree at least x.

some constant c, the technique (combined with [27]) gives a structure supporting a query in constant time and an update in $\tilde{O}(\sqrt{m} \cdot \max\{\frac{|V|^{1.5}}{\Gamma^{1.5}}, \frac{1}{\Gamma^{0.75}}\})$ time w.h.p. This bound will be strictly improved by our methods.

Lower bounds. In the *online boolean matrix-vector multiplication* (OMv) problem, an algorithm first spends poly(n) time preprocessesing an $n \times n$ boolean matrix $\boldsymbol{M}$, and is then required to compute $\boldsymbol{M}\boldsymbol{v}_i$ ($i \in [n]$) where each $\boldsymbol{v}_i$ is an $n \times 1$ boolean vector.[4] Vector $\boldsymbol{v}_{i+1}$ ($i \geq 1$) is revealed only after the algorithm has output $\boldsymbol{M}\boldsymbol{v}_i$. The *cost* is the total time spent on the n vectors.

> **OMv-conjecture [21]:** no algorithm can solve the problem with probability at least $2/3$ using *subcubic* cost $O(n^{3-\delta})$ for any constant $\delta > 0$.

The conjecture explains in a remarkable manner the computational hardness of a great variety of problems [21], and gives rise to the tight (conditional) lower bound on DETC mentioned in Section 1.1 (see [7] for the conjecture's implications on conjunctive queries when the update time has to be $\tilde{O}(1)$).

It has been shown [21] that the OMv conjecture implies another well-known conjecture formulated by Patrascu [33] on the *multiphase problem* (namely, if the former is correct, so is the latter, which means that the former is at least as hard to prove as the latter). Patrascu's conjecture has been utilized to establish (conditional) lower bounds on *dynamic set intersection emptiness* [19,28,29], which can be converted to lower bounds on DETC, but they are not tight (we will elaborate on this in Section 2). Indeed, many of the lower bounds obtained from Patrascu's conjecture can be strengthened with OMv (see [21] for a comprehensive list); the same phenomenon also applies to the DATC lower bound (Theorem 1) developed in this paper (more details in Section 2).

1.3 Our Results

DATC. Regarding Question 1 (Section 1.1), we first prove a conditional lower bound:

► **Theorem 1.** *Consider the DATC problem where $\epsilon \leq 0.49$ and $\Gamma = m^c$ for an arbitrary constant c satisfying $0 \leq c < 1/2$. Subject to the OMv-conjecture, no DATC structure can ensure $O(m^{0.5-\delta}/\Gamma)$ amortized update time and $O(m^{\frac{2}{3}-\delta})$ query time simultaneously, where $\delta > 0$ is an arbitrary constant. This is true even if the amortized update time holds only in expectation.*

We are able to match the lower bound with:

► **Theorem 2.** *There is a DATC structure that ensures $\tilde{O}((1/\epsilon)^3 \cdot \sqrt{m}/\Gamma)$ amortized update time w.h.p. and $O(1)$ query time. The space of the structure is $\tilde{O}(m + (1/\epsilon)^2 \cdot m^{1.5}/\Gamma)$.*

For constant $\epsilon \leq 0.49$, Theorems 1 and 2 together give the full tradeoff between update time and the approximation quality (subject to the OMv-conjecture). As a pleasant implication, for constant ϵ Theorem 2 shows that one can achieve $\tilde{O}(1)$ amortized update time and $O(1)$ query time by setting $\Gamma = \sqrt{m}$; in other words, we never have to worry about $\Gamma > \sqrt{m}$ (simply lower such Γ to $\sqrt{m}$). It is interesting to note, in retrospect, that the constant c in Theorem 1 does not reach $1/2$.

DETC. We address Question 2 by giving a new structure whose performance depends on the *arboricity* of G (Section 1.2):

► **Theorem 3.** *For any monotonic function $\Gamma(m)$ satisfying $1 \leq \Gamma(m) \leq \sqrt{m}$ and $\Gamma(c \cdot m) = O(\Gamma(m))$, there is a DETC structure of $O(\min\{\alpha m + m \log m, (\frac{m}{\Gamma(m)})^2\})$ space that supports an update in $\tilde{O}(\min\{\alpha + \Gamma(m), \sqrt{m}\})$ amortized time, and a query in $O(1)$ time, where α is the largest arboricity of G in history. This holds even if α is unknown.*

---

[4] Additions and multiplications are as in the boolean semi-ring.

By setting $\Gamma = \sqrt{m}$, we reconstruct the result of [27] up to polylog factors; on the other hand, we can do significantly better when α is small, i.e., G is sparse. In particular, when G is a planar graph, $\alpha = O(1)$; thus our structure achieves $O(m \log m)$ space, $\tilde{O}(1)$ amortized update time, and constant query time. The arboricity of a graph is always bounded by the h-index, but can be considerably lower, e.g., a planar graph can have an h-index of $\Theta(\sqrt{m})$; our structure is, therefore, not subsumed by [17] (Section 1.2). Similarly, even a planar graph can have a maximum vertex degree of $\Theta(|V|)$; our result is, therefore, not subsumed by [8] either. Interestingly, if α is known in advance, by setting $\Gamma = \alpha$, we obtain a structure occupying $\tilde{O}(\min\{\alpha m, m^2/\alpha^2\}) = \tilde{O}(m^{4/3})$ space that supports an update in $\tilde{O}(\alpha)$ time and ensures constant query time.

2 Hardness of Dynamic Approximate Triangle Counting

In this section, we will prove:

▶ **Lemma 4.** *Consider the DATC problem with $\epsilon = 0.49$ and $\Gamma = m^c$ for an arbitrary constant c satisfying $0 \le c < 1/2$. Subject to the OMv-conjecture, no structure can guarantee $O(m^{0.5-\delta-c})$ expected amortized update time and $O(m^{1-2c/3-\delta})$ query time, where $\delta > 0$ can be an arbitrarily small constant.*

Theorem 1 is a corollary of Lemma 4, noticing that (i) $1 - 2c/3 > 2/3$ for $c < 1/2$, and (ii) any solution that works for $\epsilon < 0.49$ must also work for $\epsilon = 0.49$. To prove the lemma, we will consider the *dynamic triangle detection* (DTD) problem, where we want to store G in a data structure to support:

- **update**(e): either adds a new edge e or removes an existing edge e;
- **query**: returns a single bit indicating whether G has any triangles at all. The query is allowed to fail with probability at most $1/m^2$.

The lemma below was first established in [21]:

▶ **Lemma 5** ([21])**.** *Subject to the OMv-conjecture, no DTD structure can guarantee $O(m^{0.5-\delta})$ amortized update time and $O(m^{1-\delta})$ query time, where $\delta > 0$ can be an arbitrarily small constant. This is true even if the amortized update time holds only in expectation.*[5]

Suppose that algorithm $\mathcal{A}$ is able to maintain a DATC structure — on our instance where $\epsilon = 0.49$ and $\Gamma = m^c$ — which supports an update in $O(m^{0.5-\delta'}/\Gamma) = O(m^{0.5-\delta'-c})$ expected amortized time and a query in $O(m^{1-2c/3-\delta'})$ time for some $\delta' > 0$. We will deploy $\mathcal{A}$ to obtain a DTD structure that contradicts Lemma 5.

Proof of Lemma 4. Henceforth, denote by G the input graph to the DTD problem, and by m the number of edges in G. Given an integer parameter $x \ge 1$, we define an *image graph* [15] G' as follows:

- for each vertex u in G, create x *image vertices* in G';
- for each edge $\{u, v\}$ in G, create x^2 *image edges* in G' by connecting every image vertex of u and every image vertex of v.

The total number of edges in G' equals $m' = x^2 m$. Observe that if G has T triangles, then the number of triangles in G' is $T' = x^3 T$.

[5] The statement in [21] (see Corollary 3.4 therein) does not contain the second sentence. Furthermore, the DTD query in [21] is not allowed to fail. However, it is easy to extend their argument to prove Lemma 5. We provide a complete proof in Appendix E.

We now proceed to explain how to support updates and DTD queries on G. For this purpose, let us first assume that $M \le m \le 2M$ for some integer $M \ge 1$. The assumption will be removed with global rebuilding, as explained later.

We choose:

$$x \quad = \quad (2M)^{\frac{c}{3-2c}}. \tag{3}$$

with which $m' = x^2 m = \Theta(m^{\frac{3}{3-2c}})$.

We apply $\mathcal{A}$ to build a DATC structure on G' (with $\epsilon = 0.49$ and $\Gamma = m'^c$). Given an **update**(e) on G, we use $\mathcal{A}$ to insert/delete all the x^2 image edges of e in G' in expected amortized time

$$O(m'^{0.5-\delta'-c} \cdot x^2) = O(m^{\frac{2c}{3-2c}+\frac{3}{3-2c}(\frac{1}{2}-\delta'-c)}) = O(m^{\frac{1}{2}-\frac{3\delta'}{3-2c}}).$$

To explain how to answer a DTD query, we will need:

▶ **Proposition 6.** $\epsilon m'^c < x^3/2$.

Proof. First note that $m' = x^2 m \le (2M)^{\frac{2c}{3-2c}} \cdot (2M) = (2M)^{\frac{3}{3-2c}}$. Hence, $\epsilon m'^c$ is at most $0.49 \cdot (2M)^{\frac{3c}{3-2c}} < x^3/2$. ◀

G has a triangle if and only if G' has at least $T' \ge x^3$ triangles. Given a DTD query on G, we run $\mathcal{A}$ to detect whether $T' \ge x^3$. For this purpose, it suffices to issue a DATC query on G'. The output t of the DATC query is greater than $x^3/2$ if and only if $T' \ge x^3$. This is because

- when $T' < x^3$, it must hold that $T' = 0$, in which case t can be at most $\epsilon \cdot \Gamma(m') = \epsilon m'^c < x^3/2$ (Proposition 6);
- when $T' \ge x^3$, $t \ge (1-\epsilon)T' \ge (1-\epsilon)x^3 > x^3/2$.

By our assumptions on $\mathcal{A}$, the DATC query runs in time

$$O(m'^{1-\frac{2c}{3}-\delta'}) = O(m^{\frac{3}{3-2c}(1-\frac{2c}{3}-\delta')}) = O(m^{1-\frac{3\delta'}{3-2c}}).$$

It remains to remove the assumption $M \le m \le 2M$. For this purpose, it suffices to destroy and rebuild the DATC structure whenever m reaches M or $2M$. The value of M for the new structure is set to $2m/3$. This makes sure $\Omega(M)$ updates on G must have happened before the next reconstruction. Standard amortization arguments show that the amortized update time is still $O(m^{\frac{1}{2}-\frac{3\delta'}{3-2c}})$ in expectation.

We thus have obtained a DTD structure with expected amortized update time $O(m^{0.5-\delta})$ and query time $O(m^{1-\delta})$ with $\delta = \frac{3\delta'}{3-2c}$, contradicting Lemma 5. This completes the proof of Lemma 4.

Remarks. A weaker lower bound would result from Patrascu's multiphase conjecture [33]. Consider, for simplicity, $c = 0$ (essentially, exact counting) in which case the strongest lower bound derived with that conjecture [28, 29] asserts that no structure can guarantee $O(m^{1/3-\delta})$ update and query time simultaneously[6]. This is also the best we can prove by executing our argument on the multiphase conjecture, but is worse than Theorem 1 by a polynomial factor. Finally, it is worth mentioning that our argument actually works for any $\epsilon < 0.5$.

3 A Structure for Dynamic Approximate Triangle Counting

This section presents a DATC structure which achieves the performance in Theorem 2.

6 A DETC structure with $O(m^{1/3-\delta})$ update and query time will lead to $t_i = O(N^{1/3-\delta})$ and $t_q = O(N^{1/3-\delta})$ in the context of Theorem 9 of [28], causing a contradiction there.

3.1 Overview

We will start by describing a "folklore" algorithm (see Section 3.6 for a discussion) for approximate triangle counting on a *static* graph $G = (V, E)$. Denote by $d(u)$ the degree of vertex $u \in V$. Define an ordering $\prec$ on V: $u \prec v$ if $d(u) < d(v)$, breaking ties by id. Orient G by pointing each edge $\{u, v\} \in E$ from u to v where $u \prec v$. Let E^+ be the set of directed edges thus obtained, and define $G^+ = (V, E^+)$ as the resulting directed graph. Denote by $d^+(u)$ the out-degree of $u \in V$ in G^+; it must hold that $d^+(u) = O(\sqrt{m})$.

To estimate the number T of triangles, initialize $\Lambda = 0$, and repeat the following $s = \tilde{O}((1/\epsilon)^2 \cdot m^{1.5}/T)$ times:

1. Take an edge $(u, v) \in E^+$ and then an out-neighbor w of u, both uniformly at random (note that v may be w). We will refer to (u, v, w) as a *random tuple*.
2. Add the *contribution* of (u, v, w) to Λ, which is $d^+(u)$ if $(v, w) \in E^+$, or 0 otherwise.

Finally, return $\Lambda \cdot (m/s)$ as the estimate, guaranteed to enjoy a relative error at most ϵ w.h.p.

Our structure dynamizes the above algorithm, as outlined next.

Standard ideas. We

- replace T with Γ (Section 1), and
- maintain a set S of $s = \tilde{O}((1/\epsilon)^2 \cdot m^{1.5}/\Gamma)$ random tuples, as well as the sum Λ of their contributions.

Inserting/deleting an edge $\{u, v\}$ may flip the directions of many edges, rendering it expensive to keep G^+ up-to-date. But the issue can be easily remedied: it suffices to flip an edge only after $\Omega(\min\{d(u), d(v)\})$ updates. For this purpose, we introduce a function D such that $D(u)$ approximates $d(u)$ up to a small constant factor for every $u \in V$. Accordingly, $\prec$ is redefined with respect to D: $u \prec v$ if $D(u) < D(v)$, breaking ties by id. We can then afford to materialize G^+ explicitly by updating it only when D changes.

$D(u)$ is adjusted when it ceases to approximate $d(u)$. When this happens, some edges of u in G^+ have their directions flipped, e.g., (u, v) becomes (v, u). A major challenge now enters the picture: *the altering of $d^+(v)$ may affect all the contributions of the random tuples (x, y, z) with $x = v$!* Specifically, each $(v, y, z) \in S$ may have already registered in Λ a contribution $d^+(v)$, which therefore must be modified. Unfortunately, we cannot afford to do so for all neighbors v of u.

New ideas. We overcome the above challenge by introducing another function D^+ such that $D^+(u)$ approximates $d^+(u)$ up to some small factor for every $u \in V$. For each random tuple $(u, v, w) \in S$, its contribution is either $D^+(u)$ — as opposed to $d^+(u)$ — or 0. Only when $D^+(u)$ ceases to approximate $d^+(u)$ will we adjust the tuple's contribution in Λ. This "two-level approximation" (i.e., D and D^+) turns out to be the key in our solution to DATC. We will argue that D, D^+, S, and Λ can be maintained efficiently along with the edge updates.

3.2 Structure

Our discussion will assume that the number m of edges in G satisfies $M \le m \le 2M$ for some integer $M \ge 1$. The assumption can be removed by reconstructing our structure periodically.

Main structure. Let $D : V \to \mathbb{N}$ be a function such that for every $u \in V$:

$$D(u) \begin{cases} = 2 & \text{if } d(u) \le 1 \\ \in [\frac{1}{2}d(u), \frac{3}{2}d(u)] & \text{otherwise.} \end{cases} \tag{4}$$

As mentioned, for two distinct vertices $u, v \in V$, $u \prec v$ if $D(u) < D(v)$, breaking ties by id. This gives rise to the directed graph $G^+ = (V, E^+)$ as defined in Section 3.1. Let $D^+ : V \to \mathbb{N}$ be another function such that for every $u \in V$:

$$D^+(u) \quad \in \quad \Big[(1-\epsilon/2)\cdot d^+(u), (1+\epsilon/2)\cdot d^+(u)\Big]. \tag{5}$$

During an edge insertion/deletion, function D (or D^+, resp.) may temporarily violate (4) (or (5), resp.), in which case we say that the function is *bad*. D (or D^+, resp.) is *good* when no violation occurs. At the beginning or right after reconstruction, $D^+(u) = d^+(u)$ for all $u \in V$; and $D(u) = d(u)$ if $d(u) \geq 2$, or 2 otherwise.

Set $s = \tilde{O}((1/\epsilon)^2 \cdot M^{1.5}/\Gamma(M))$; note that the function $\Gamma(.)$ is parameterized for the smallest possible $m = M$. Define S to be a set of s independent random tuples drawn from G^+ (Section 3.1). Each tuple $(x, y, z) \in S$ makes a *contribution*

$$f(x,y,z) \quad = \quad \begin{cases} D^+(x) & \text{if } (y,z) \in E^+ \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Set

$$\Lambda \quad = \quad \sum_{(x,y,z)\in S} f(x,y,z). \tag{7}$$

Given vertices $u, v \in V$, define:

$$\Xi_{u,v} \quad = \quad \sum_{(x,u,v)\in S} D^+(x) \tag{8}$$

where the summation is over the random tuples (x, y, z) satisfying $y = u, z = v$. The pair (u, v) is *active* if at least one such random tuple exists.

Our structure can be summarized as:

- graphs G and G^+
- functions D and D^+
- the set S of random tuples
- the value of Λ, and values of $\Xi_{u,v}$'s for all active (u, v).

It is worth pointing out that Λ and the $\Xi_{u,v}$'s do *not* imply the need to maintain the contribution function f in (6).

Filtered subsets of S. We will use "$\perp$" to denote a wildcard, and define the boolean expression "$u = \perp$" to be true for any $u \in V$. Given q_1, q_2, and q_3 where each q_i ($1 \leq i \leq 3$) is either a vertex or a wildcard, we introduce:

$$S_{q_1,q_2,q_3} \quad = \quad \{(x,y,z) \in S \mid x = q_1, y = q_2, z = q_3\}$$

namely, the subset obtained by filtering S using q_1, q_2, q_3.

▶ **Lemma 7.** *All the statements below are true:*

- *For any $u \in V$, $|S_{u,\perp,\perp}| = \tilde{O}(d^+(u) \cdot s/m)$ w.h.p.*
- *For any $u, v \in V$ such that $(u, v) \in E^+$, $|S_{u,v,\perp}| = \tilde{O}(s/m)$ w.h.p.*
- *For any $u, v \in V$ such that $(u, v) \in E^+$, $|S_{u,\perp,v}| = \tilde{O}(s/m)$ w.h.p.*

Proof. A random tuple (x, y, z) satisfies $x = u$ if and only if (x, y) is an out-edge of u in G^+. As (x, y) is a random edge in G^+, it is an out-edge of u with probability $d^+(u)/m$. Due to independence, $|S_{u,\perp,\perp}|$ is $\tilde{O}(s \cdot d^+(u)/m)$ w.h.p., as stated in the first bullet.

To prove the 2nd (or 3rd, resp.) bullet, it suffices to show that (x, y, z) belongs to $|S_{u,v,\perp}|$ (or $|S_{u,\perp,v}|$, resp.) with probability $1/m$. This is obvious for $S_{u,v,\perp}$. For (x, y, z) to appear in $S_{u,\perp,v}$:

- (x, y) must be an out-edge of u, which happens with probability $d^+(u)/m$;

- z chooses v, which happens with probability $1/d^+(u)$.

Therefore, $\mathbf{Pr}[(x, y, z) \in S_{u,\perp,v}] = 1/m$. ◀

Auxiliary structures. We assume the availability of *auxiliary structures* for:

- Given any q_1, q_2 and q_3, retrieve the size of S_{q_1,q_2,q_3} in $\tilde{O}(1)$ time.
- Given any q_1, q_2, q_3 and an integer k between 1 and $|S_{q_1,q_2,q_3}|$, uniformly sample k tuples *without replacement* (WoR) from S_{q_1,q_2,q_3} in $\tilde{O}(k)$ time. By setting $k = |S_{q_1,q_2,q_3}|$, we can use the operation to extract the entire S_{q_1,q_2,q_3}.
- Given any $u, v \in V$, in $\tilde{O}(1)$ time either retrieve $\Xi_{u,v}$ or assert that (u, v) is not active.
- Generate a random tuple from G^+ in $\tilde{O}(1)$ time.

All the auxiliary structures can be implemented as simple variants of binary search trees (see Chapter 14 of [14]).

Space. The overall space consumption is clearly $O(m + s) = \tilde{O}(m + (1/\epsilon)^2 \cdot m^{1.5}/\Gamma(m))$, using the fact that $\Gamma(m) \le \Gamma(2M) = O(\Gamma(M))$.

Query. We will prove in Appendix B:

▶ **Lemma 8.** *With probability at least $1 - 1/m^3$, the value $\Lambda \cdot (M/s)$ is an estimate satisfying the $(\epsilon, \Gamma(m))$ guarantee.*

A query can therefore be answered in constant time.

Remarks. The following subsections will explain how to support insertions. The deletion algorithm is similar, with details duly presented in Appendix C.

Our discussion will ignore the auxiliary structures because they are rudimentary; and their maintenance cost can be higher than that of S and $\{\Xi_{u,v} \mid \text{active } (u, v)\}$ by at most a logarithmic factor. Furthermore, when a tuple (x, y, z) is inserted/deleted in S, Λ and $\Xi_{y,z}$ can be updated accordingly in logarithmic time. We will, therefore, not discuss explicitly the modifications to Λ and $\{\Xi_{u,v} \mid \text{active } (u, v)\}$ caused by insertions/deletions in S.

3.3 Insertion: When D Will Still Be Good

Suppose that we are inserting an edge $\{u^*, v^*\}$ in G. After the insertion, $d(u^*)$ and $d(v^*)$ both increase by 1. In this section, we consider that *D is still good for the new $d(u^*)$ and $d(v^*)$*. Consequently, every existing edge in G^+ retains its direction. Without loss of generality, assume that $\{u^*, v^*\}$ points from u^* to v^* in G^+.

Rationale. How would this affect a random tuple $(x, y, z) \in S$? Recall that (x, y) is supposed to be drawn uniformly at random from E^+. Now that m has increased by 1, (x, y) should be replaced by (u^*, v^*) with probability $1/m$ (reservoir sampling [38]). If the replacement occurs, (x, y, z) is said to be *edge-replaced*; in this case, we take a (uniformly) random out-neighbor w of u^*, delete (x, y, z) from S, and add (u^*, v^*, w).

For a tuple (x, y, z) that is *not* edge-replaced, further processing is necessary in two cases:

- Case 1: $x = u^*$. Since u^* has got a new out-neighbor v^*, z (which is supposedly a random out-neighbor of x) should be replaced by v^* with probability $1/d^+(u^*)$. If the replacement happens, (x, y, z) is said to be *outneighbor-replaced*; in this case, we delete (x, y, z) from S and add (u^*, y, v^*) instead.
- Case 2: $y = u^*, z = v^*$. The new edge (u^*, v^*) completes the triangle formed by x, u^*, v^*. We should therefore increase Λ (see (7)) by $f(x, y, z) = D^+(x)$.

Insertion algorithm. Figure 1 presents the algorithm in pesudocode. To find the edge-replaced tuples, we cannot afford to toss a coin for each tuple in S. However, we do not have to; because

algorithm insert (u^*, v^*) /* a new edge (u^*, v^*) has just been added to G^+ */

1. generate an integer k_1 following the binomial distribution $B(|S|, 1/m)$
2. $S_1 \leftarrow$ a size-k_1 WoR sample set of S; remove S_1 from S
3. generate an integer k_2 following the binomial distribution $B(|S_{u^*,\perp,\perp}|, 1/d^+(u^*))$
4. $S_2 \leftarrow$ a size-k_2 WoR sample set of $S_{u^*,\perp,\perp}$; remove S_2 from S
   /* the removal of each $(x, y, z) \in S_1 \cup S_2$ requires updating Λ and $\Xi_{y,z}$ */
5. increase Λ by Ξ_{u^*,v^*}
6. **repeat** k_1 **times**
7.   add (u^*, v^*, w) into S where w is a (uniformly) random out-neighbor of u^*
       /* requires updating Λ and $\Xi_{v^*,w}$ */
8. **for** each $(u^*, y, z) \in S_2$ **do**
9.   add (u^*, y, v^*) to S /* requires updating Λ and Ξ_{y,v^*} */

Figure 1 Pseudocode of the insertion algorithm

the tuples in S are independent, it suffices *generate* how many — say k_1 — edge-replaced tuples there should be, and draw a WoR sample set S_1 of size k_1 from S. Here, k_1 follows the binomial distribution $B(|S|, 1/m)$, and can be generated in $\tilde{O}(1)$ time (see, e.g., [38]). Using the auxiliary structures, we can extract S_1 and remove the tuples therein from S (Lines 1-2) in $\tilde{O}(k_1)$ time where $k_1 = \tilde{O}(|S|/m) = \tilde{O}(s/m)$ w.h.p. The same idea also applies to outneighbor-replaced tuples in Case 1. The number k_2 of such tuples follows the binomial distribution $B(|S_{u^*,\perp,\perp}|, \frac{1}{d^+(u^*)})$; hence, $k_2 = \tilde{O}(|S_{u^*,\perp,\perp}|/d^+(u^*)) = \tilde{O}(s/m)$ w.h.p. (Lemma 7). From $S_{u^*,\perp,\perp}$, we extract a WoR sample set S_2 of size k_2 in $\tilde{O}(k_2) = \tilde{O}(s/m)$ time using the auxiliary structures; S_2 can be regarded as the set of outneighbor-replaced tuples, which are then removed from S in $\tilde{O}(s/m)$ time (Line 3-4). Increasing the value of Λ due to Case 2 can be accomplished by simply adding Ξ_{u^*,v^*} (defined in (8)) to Λ (Line 5). The value of Ξ_{u^*,v^*} can be retrieved in $\tilde{O}(1)$ time from the auxiliary structures. Lines 6-9 then replenish S for the random tuples in $S_1 \cup S_2$ removed earlier.

After the insertion, the out-degree $d^+(u^*)$ of u^* increases by 1. If $D^+(u^*)$ still satisfies (5), the insertion is complete. Otherwise, we call **fix-Dplus**(u^*) (introduced below) and finish. In summary, the insertion runs in $\tilde{O}(s/m)$ time, plus the cost of **fix-Dplus**(u^*).

Algorithm fix-Dplus(u). This algorithm has the following constraint:

Invariant: when called, $D^+(u)$ violates (5).

fix-Dplus(u) first makes a copy of the current $D^+(u)$ — denote the copy as $\mathcal{D}^+_{old}$ — and then resets $D^+(u)$ to $d^+(u)$. Accordingly, for every $(x, y, z) \in S$ with $x = u$, its contribution $f(x, y, z)$ may change from $\mathcal{D}^+_{old}$ to $d^+(u)$. This may affect Λ and every $\Xi_{v,w}$ where v and w are out-neighbors of u in G^+. To remedy all these, we first retrieve $S_{u,\perp,\perp}$, and then for every $(u, y, z) \in S_{u,\perp,\perp}$:

- if $(y, z) \in E^+$, increase Λ by $d^+(u) - \mathcal{D}^+_{old}$;
- increase $\Xi_{y,z}$ by $d^+(u) - \mathcal{D}^+_{old}$.

By Lemma 7, $S_{u,\perp,\perp} = \tilde{O}(d^+(u) \cdot s/m)$ w.h.p. This implies:

▶ **Lemma 9.** *The cost of* **fix-Dplus**(u) *is* $\tilde{O}(|\mathcal{D}^+_{old} - d^+(u)| \cdot s/(\epsilon m))$ *w.h.p.*

Proof. The cost of **fix-Dplus**(u) is $\tilde{O}(d^+(u) \cdot s/m)$. Next, we show $d^+(u) = O(|\mathcal{D}^+_{old} - d^+(u)|/\epsilon)$. Consider the two possibilities of how $D^+(u)$ can violate (5). If $\mathcal{D}^+_{old} > (1+\epsilon/2) \cdot d^+(u)$, then $d^+(u) < (\mathcal{D}^+_{old} - d^+(u)) \cdot (2/\epsilon)$. On the other hand, if $\mathcal{D}^+_{old} < (1-\epsilon/2) \cdot d^+(u)$, we have $d^+(u) < (d^+(u) - \mathcal{D}^+_{old}) \cdot (2/\epsilon)$. ◀

3.4 Insertion: When D Will Go Bad

Again, denote by $\{u^*, v^*\}$ the edge to be inserted. This time, we consider that D will be bad after $d(u^*)$ and $d(v^*)$ increase by 1. In other words, D will cease to satisfy (4) with respect to u^*, v^*, or both. Our strategy is *not* to perform the insertion immediately. Instead, we will first modify D to make sure that D will *still* be good after the insertion. Then, the insertion can be processed by the algorithm in Section 3.3.

Next, we will introduce an algorithm named **fix-D** which takes a vertex u as the parameter, and has the following constraint:

Invariant: when called:
- D is good
- $D(u) < d(u)$ and $d(u) = O(D(u))$, and
- $d(u) - D(u) = \Omega(D(u))$.

At the end of **fix-D**(u), $D(u) = d(u)$, which ensures that $D(u)$ will still satisfy (4) even after $d(u)$ grows by 1. Thus, for the aforementioned insertion, we can simply invoke **fix-D**(u^*) and/or **fix-D**(v^*), depending on which will cause D to go bad.

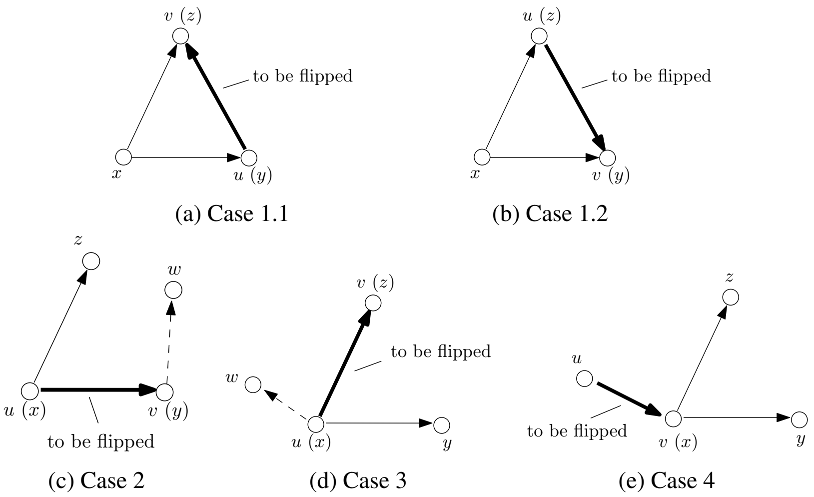
Rationale behind fix-D(u). We increase $D(u)$ to $d(u)$. Recall that, for each neighbor v of u in G, the edge $\{u, v\}$ is given a direction in G^+. The increase of $D(u)$ may affect the direction: if the direction was (u, v) before, it may now be flipped to (v, u); on the other hand, if the direction was (v, u), it remains the same.

The direction flipping can invalidate S because a tuple in S may stop being a *random* tuple, or its contribution as in (6) may change (which will further affect Λ). To explain, fix a tuple $(x, y, z) \in S$, and suppose that an edge (u, v) is to be flipped to (v, u). Next, we enumerate all possible cases where modifications are necessary:

- Case 1: $x \neq u$ and $x \neq v$. (x, y, z) will remain as a random tuple. However, its contribution $f(x, y, z)$ is affected in two subcases:
  - Case 1.1: $y = u$ and $z = v$. $f(x, y, z)$ will drop from $D^+(x)$ to 0. Accordingly, Λ needs to be decreased by $D^+(x)$. See Figure 2(a).
  - Case 1.2: $y = v$ and $z = u$. $f(x, y, z)$ will grow from 0 to $D^+(x)$. Accordingly, Λ needs to be increased by $D^+(x)$. See Figure 2(b).
- Case 2: $x = u$ and $y = v$. (x, y, z) will become invalid due to the disappearance of (x, y). The tuple (u, v, z) should be replaced by (v, u, w) where w is a (uniformly) random out-neighbor of v. See Figure 2(c).
- Case 3: $x = u$, $y \neq v$, and $z = v$. (x, y, z) will become invalid due to the disappearance of (x, z). The tuple (u, y, v) should be replaced by (u, y, w) where w is a (uniformly) random out-neighbor of u. See Figure 2(d).
- Case 4: $x = v$ (which implies $y \neq u$ and $z \neq u$). Since v has gained a new out-neighbor u, (x, y, z) may no longer be random. To remedy this, z should be replaced by u with probability $1/d^+(v)$. If the replacement occurs, the tuple (v, y, z) is said to be *outneighbor-replaced*. See Figure 2(e).

Algorithm fix-D(u). We start by setting $D(u) = d(u)$, flipping the edges of u in G^+ wherever needed.

Given each neighbor v of u in G such that $\{u, v\}$ was flipped, we
- (for Case 1) retrieve $\Xi_{u,v}$ and $\Xi_{v,u}$ (from the auxiliary structures), and increase Λ by $\Xi_{v,u} - \Xi_{u,v}$.
- (for Case 2) retrieve $S_{u,v,\perp}$; and then for each $(u, v, z) \in S_{u,v,\perp}$, delete (u, v, z) from S, pick an out-neighbor w of v uniformly at random, and add (v, u, w) to S.



(a) Case 1.1 (b) Case 1.2

(c) Case 2 (d) Case 3 (e) Case 4

Figure 2 Different cases of **fix-D**

- (for Case 3) retrieve $S_{u,\perp,v}$; and then for each $(u, y, v) \in S_{u,\perp,v}$ with $y \neq v$, delete (u, y, v) from S, pick an out-neighbor w of u uniformly at random, and add (u, y, w) to S.

By Lemma 7, $S_{u,v,\perp}$ and $S_{u,\perp,v}$ both have size $\tilde{O}(s/m)$ w.h.p. Thus, Cases 1-3 can be handled in $\tilde{O}(d(u) \cdot s/m))$ time w.h.p.

Next, we focus on Case 4. Let v be a neighbor of u with $\{u, v\}$ flipped. The number k_v of outneighbor-replaced tuples (x, y, z) with $x = v$ follows the binomial distribution $B(|S_{v,\perp,\perp}|, 1/d^+(v))$. Combining this with (the first bullet of) Lemma 7 shows that $k_v = \tilde{O}(d^+(v) \cdot \frac{s}{m} \cdot \frac{1}{d^+(v)}) = \tilde{O}(s/m)$ w.h.p. We extract a WoR sample set of size k_v from $S_{v,\perp,\perp}$[7], which takes $\tilde{O}(k_v) = \tilde{O}(s/m)$ time using the auxiliary structures. Every tuple (v, y, z) extracted is then modified to (v, y, u) in $\tilde{O}(1)$ time. Therefore, the total cost of Case 4 is again $\tilde{O}(d(u) \cdot s/m)$ w.h.p.

Now, let us worry about the function D^+. Compared to before **fix-D**(u) was called, $d^+(u)$ may have changed abruptly (by as much as $d(u)$ in the worst case). If $D^+(u)$ now violates (5), we invoke **fix-Dplus**(u). Finally, for each neighbor v of u in G, $d^+(v)$ may have changed by 1, compared to before **fix-D**(u) was called. $D^+(v)$ may no longer satisfy (5); if so, call **fix-Dplus**(v).

In summary, **fix-D**(u) runs in $\tilde{O}(d(u) \cdot s/m)$ time w.h.p., plus the cost of all the calls to **fix-Dplus** at the end. It is worth pointing out that the invariant of **fix-D**(u) ensures $d(u) = O(\mathcal{D}_{old})$, where $\mathcal{D}_{old}$ is the value of $D(u)$ at the beginning of **fix-D**(u).

3.5 Analysis

Section 3.3 has shown that an insertion finishes in $\tilde{O}(s/m)$ time w.h.p. if no calls to **fix-Dplus** or **fix-D** are made. It remains to discuss the time spent on **fix-Dplus** and **fix-D**.

Let us start with **fix-D**. Consider its execution on a node u. Denote by $\mathcal{D}_{old}$ the value of $D(u)$ at the beginning of **fix-D**(u). Recall that **fix-D**(u) has cost $\tilde{O}(\mathcal{D}_{old} \cdot s/m)$ w.h.p., plus the cost of some calls to **fix-Dplus** at the end. We will account for the $\tilde{O}(\mathcal{D}_{old} \cdot s/m)$ cost first, and worry about **fix-Dplus** later. The invariant of **fix-D** (Section 3.4) makes sure that $\Omega(\mathcal{D}_{old})$ edges incident on u must have been inserted since the last time **fix-D** was invoked on u. We can therefore charge the $\tilde{O}(\mathcal{D}_{old} \cdot s/m)$ cost over those insertions, each of which bears only $\tilde{O}(s/m)$.

Let us now turn attention to **fix-Dplus**, for which we use a token-based analysis. A *token* is generated in two scenarios:

[7] Precisely speaking, this should be the $S_{v,\perp,\perp}$ at the beginning of **fix-D**(u).

- Case 1: in Section 3.3, when an edge (u^*, v^*) is added to G^+, we give a token to u^* because its out-degree will increase by 1.
- Case 2: during the execution of **`fix-D`**(u), when we flip an in/out-edge of u with respect to an in/out-neighbor v, we give both u and v a token because their out-degrees will change by 1.

▶ **Lemma 10.** *If the total number of edge insertions is n_{ins}, the number of tokens generated is $O(n_{ins})$.*

Proof. The number of tokens in Case 1 is clearly n_{ins}. Next, we focus on Case 2. Let $\mathcal{D}_{old}$ be the value of $D(u)$ at the beginning of **`fix-D`**(u). Case 2 can generate at most $2d(u)$ tokens, while $2d(u)$ is $O(\mathcal{D}_{old})$ due to the invariant of **`fix-D`**. As mentioned, $\Omega(\mathcal{D}_{old})$ edges incident on u must have been inserted since the last **`fix-D`**(u). Thus, after amortization, each of those insertions generates $O(1)$ tokens in Case 2. ◀

Consider a call to **`fix-Dplus`**(u). Let $\mathcal{D}^+_{old}$ be the value of $D^+(u)$ at the beginning of the call. Clearly, u must have received at least $|\mathcal{D}^+_{old} - d^+(u)|$ tokens since the last **`fix-Dplus`**(u). We can charge the cost $\tilde{O}(|\mathcal{D}^+_{old} - d^+(u)| \cdot s/(\epsilon m))$ of **`fix-Dplus`**(u) over those tokens, each of which is amortized only $\tilde{O}(s/(\epsilon m))$. Combined with Lemma 10, this means that each insertion is amortized a share of $\tilde{O}(s/(\epsilon m))$.

In summary, each insertion runs in $\tilde{O}(s/(\epsilon m)) = \tilde{O}((1/\epsilon)^3 \cdot \sqrt{m}/\Gamma)$ amortized time w.h.p. This, together with the deletion algorithm in Appendix C, establishes Theorem 2.

3.6 Discussion

There is a rich literature on approximate triangle counting; for entry points into the literature, see [2–5, 9–12, 15, 18, 23–26, 30–32, 34, 35, 37]. The presented data structure reflects our efforts in identifying the existing techniques suitable for DACT. Strictly speaking, the "folklore" static-counting algorithm in Section 3.1 has not been formally documented; however, its underlying ideas are already known. First, orienting the edges in the way described is a standard approach (e.g., [2, 4, 13, 16, 22, 31]). Second, the sampling procedure for acquiring "random tuples" is commonly known as *wedge sampling*, and is an important method behind many algorithms (e.g., [2, 4, 10, 16, 18, 23, 31, 34, 35]). Third, the notion of *contribution* (defined in (6)) is what makes wedge sampling work in our context, and was inspired by a subroutine inside an algorithm developed in [16] (see the `Heavy` subroutine therein). Our contributions, on the other hand, are in maintaining the information needed by the static algorithm under updates. The two-level approximation idea — manifested by the functions D and D^+ — is unlikely the only way to make things work, but has helped considerably in making our arguments as clean as possible.

4 A Structure for Dynamic Exact Triangle Counting

This section presents a DETC structure that achieves the performance in Theorem 3. Our algorithms and analysis can be regarded as a fine-grained version of those in [27].

4.1 Structure

We assume that the number m of edges in $G = (V, E)$ satisfies $M \le m \le 2M$ for some integer $M \ge 1$; the assumption can be removed by standard global rebuilding. As stated in Theorem 3, our structure takes a function $\Gamma(.)$ as a parameter. Set $\lambda = \Gamma(M)$ in the following discussion.

Graph orientation. At any moment, we orient G by giving each edge $\{u, v\}$ in G a direction. Let E^+ be the set of directed edges obtained, and denote by $G^+ = (V, E^+)$ the resulting directed graph. Denote by $d^+(u)$ the out-degree of $u \in V$. The orientation is done according to:

▶ **Lemma 11** ([6]). *By spending $O(\log m)$ worst-case time on an (edge) insertion/deletion in G, we can maintain G^+ such that $d^+(u) = O(\alpha + \log m)$ for every $u \in V$, where α is the largest arboricity of G in history. Furthermore, each insertion/deletion in G flips the directions of $O(\log m)$ edges in G^+. The above statements are true even if α is unknown.*

Since $M \le m \le 2M$ holds at all times, we must have $\alpha = O(\sqrt{M}) = O(\sqrt{m})$. Note that G^+ *can contain cycles* (it differs from the G^+ in Section 3.2). For each $u \in V$, denote by $N^+(u)$ the set of out-neighbors of u, and by $N^-(u)$ the set of its in-neighbors.

Light, heavy, and active H-combos. We classify each vertex $u \in V$ as *light* or *heavy* based on its degree $d(u)$ in G according to the rules below:

- if $d(u) \le \lambda/2$, always light, whereas if $d(u) \ge \lambda$, always heavy;
- when our data structure is just constructed, u is heavy if $d(u) \ge 3\lambda/4$ or light otherwise;
- if u is heavy, it switches to light only when $d(u)$ has dropped to $\lambda/2$;
- if u is light, it switches to heavy only when $d(u)$ has increased to λ.

Given two distinct heavy vertices $u, v \in V$, define:

$$I_{\{u,v\}} \quad = \quad |N^-(u) \cap N^-(v)|$$

namely, the number of their common in-neighbors in G^+. $\{u, v\}$ forms an *active H-combo* if $I_{\{u,v\}} > 0$; note that there may not be an edge between u and v in G. Notice that while light/heavy-vertices are defined based on G, $I_{\{u,v\}}$ is defined based on G^+. This is a crucial design to attain the performance in Theorem 3.

Structure. We maintain

- G and G^+
- the number T of triangles in G
- the set A of active H-combos, and $\mathcal{I}_A = \{I_{\{u,v\}} \mid \{u, v\} \in A\}$.

We also assume *auxiliary structures* for:

- given any heavy vertices u, v, finding $I_{\{u,v\}}$ in $\tilde{O}(1)$ time or declaring that $\{u, v\} \notin A$;
- inserting, deleting, or modifying an element in $\mathcal{I}_A$ using $\tilde{O}(1)$ time.

▶ **Lemma 12.** *The above structure consumes $O(\min\{\alpha m + m \log m, (m/\lambda)^2\})$ space.*

Proof. The auxiliary structures only need to be binary search trees which consume $O(|A|)$ space. It suffices to bound the size of A. Note that the number of heavy vertices is $O(m/\lambda)$ which immediately implies $|A| = O((m/\lambda)^2)$. Next, we will prove that $|A|$ is also bounded by $O(\alpha m + m \log m)$. Remember that each active H-combo $\{u, v\}$ must have a common in-neighbor. Conversely, each vertex $w \in V$ can generate $O(|N^+(w)|^2)$ active H-combos. By Lemma 11, $|N^+(w)| = O(\alpha + \log m)$. Therefore, $|A| = O(\sum_{w \in V} |N^+(w)|^2) = O(\sum_{w \in V} |N^+(w)| \cdot (\alpha + \log m)) = O(m(\alpha + \log m))$. ◀

Each (DETC) query obviously can be answered in constant time.

Remarks. For each edge update in G, Lemma 11 flips $O(\log m)$ edges in G^+. We implement the flipping of an edge (u, v) by first deleting (u, v) from G^+ and then adding (v, u) back. In this way, the number of edge updates on G^+ can be higher than that on G by at most a logarithmic factor. Thus, it suffices to discuss how to add/remove a (directed) edge in G^+.

Next, we will explain how to support insertions. The deletion algorithm is similar and thus moved to Appendix D. Our discussion will ignore the auxiliary structures. Furthermore, whenever an H-combo $\{u, v\}$ is inserted/deleted in A, $I_{\{u,v\}}$ can be inserted/deleted accordingly in logarithmic time. We will therefore not elaborate on the modifications to $\mathcal{I}_A$ caused by insertions/deletions in A.

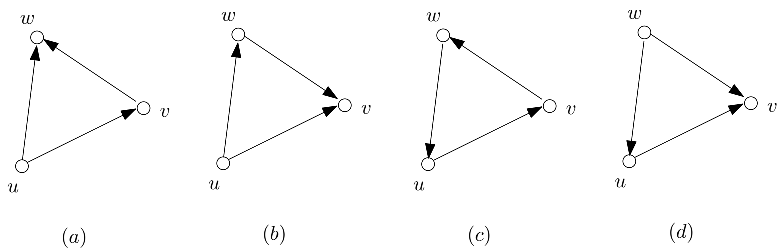


Figure 3 Four different triangles to be counted

4.2 Insertion

Update T. Given a new edge (u, v) in G^+, Figure 3 shows the possible cases for a triangle involving u and v, in terms of the edge directions. The types in Figures 3(a), 3(b), and 3(c) have an out-edge of u, v, or both, and hence, can be *enumerated* directly by scanning through the out-neighbors of u and v. The time required is $\tilde{O}(d^+(u) + d^+(v)) = \tilde{O}(\alpha)$ by Lemma 11.

Regarding Figure 3(d), we distinguish two cases:

- Case 1: u or v is a light vertex. If u (or v, resp.) is a light vertex, go through its $O(\lambda)$ in-edges to enumerate triangles of Figure 3(d) in $\tilde{O}(\lambda)$ time.
- Case 2: u and v are both heavy vertices. The number of such triangles is $I_{\{u,v\}}$, and can be retrieved from the auxiliary structures in $\tilde{O}(1)$ time.

Therefore, T can be updated in $\tilde{O}(\alpha + \lambda)$ time.

Update $\mathcal{I}_A$ and A. If v is heavy, every heavy out-neighbor w of u (other than v) forms an active H-combo with v. If $\{v, w\}$ is already in A, increase $I_{\{v,w\}}$ by 1; otherwise, add $\{v, w\}$ to A. This requires $\tilde{O}(d^+(u)) = \tilde{O}(\alpha)$ time in total.

Now, u and/or v may have just turned from light to heavy. It suffices to concentrate on u due to symmetry. We examine every in-neighbor x of u in G. For each heavy out-neighbor y of x ($y \neq u$), either add $\{u, y\}$ to A or increase $I_{\{u,y\}}$ by 1. The total time is $\tilde{O}(\alpha\lambda)$ because u has at most λ in-neighbors, each having an out-degree $\tilde{O}(\alpha)$. We charge the time on the $\Omega(\lambda)$ edges of u that have been added since u turned light last time; the insertion of each of those edges bears only $\tilde{O}(\alpha)$ time.

Combining the above with the deletion algorithm in Appendix D establishes Theorem 3.

5 Conclusions

Triangle counting is an important problem with numerous applications in database systems. Recent research on this topic has looked at how to maintain the number of triangles as edges are inserted and deleted in the underlying graph. Unfortunately, the exact triangle count is expensive to maintain in the worst case because it requires $\Omega(\sqrt{m})$ time per update where m is the number of edges (subject to a widely accepted conjecture). In this paper, we seek to reduce the update overhead with two orthogonal approaches. The first one introduces imprecision in counting and aims to strike an optimal tradeoff between the update cost and the permissible error. The second approach still does exact counting, but aims to bound the update time using arboricity, which is an intrinsic parameter of the input graph for characterizing its sparsity. Our contributions include data structures and algorithms with non-trivial performance bounds in each of the two directions, and a matching (conditional) lower bound in the first direction.

Appendix

A Chernoff Bounds

Let $\mathcal{X}_1, ..., \mathcal{X}_n$ be independent random variables between 0 and 1. If $\mathcal{X} = \sum_{i=1}^{n} \mathcal{X}_i$ and $\mu = \mathbf{E}[\mathcal{X}]$, then for any $0 \le \gamma \le 1$:

$$\mathbf{Pr}\big[|\mathcal{X} - \mu| \ge \gamma \cdot \mu\big] \quad \le \quad 2\exp\left(-\frac{\gamma^2 \mu}{3}\right) \tag{9}$$

and for any $\gamma \ge 1$:

$$\mathbf{Pr}[\mathcal{X} \ge (1+\gamma) \cdot \mu] \quad \le \quad \exp\left(-\frac{(1+\gamma)\mu}{6}\right). \tag{10}$$

These bounds can be found in [36].

B Proof of Lemma 8

We will use $N^+(u)$ to represent the set of out-neighbors of u in G^+.

▶ **Lemma 13.** *For every vertex $u \in V$, $d^+(u) \le \max\{4, \sqrt{6m}\}$.*

Proof. We consider only $d(u) > 4$ because otherwise the claim obviously holds. For each out-neighbor v of u in G^+, its degree $d(v)$ in G must be at least 2. To see this, suppose on the contrary $d(v) \le 1$, which implies $D(v) = 2 < \frac{d(u)}{2} \le D(u)$ (the last $\le$ is due to (4)). This means that the edge $\{u, v\}$ should point from v to u, giving a contradiction.

By (4), the fact $d(v) \ge 2$ indicates $D(v) \le \frac{3}{2}d(v)$. We now have $\frac{d(u)}{2} \le D(u) \le D(v) \le \frac{3d(v)}{2}$, namely, $d(u) \le 3d(v)$. It follows that

$$d^+(u)^2 \le d^+(u) \cdot d(u) = \sum_{v \in N^+(u)} d(u) \le \sum_{v \in N^+(u)} 3d(v) \le 6m$$

thus completing the proof. ◀

Let us introduce

$$D^+_{max} \quad = \quad (1 + \epsilon/2) \cdot \max\{4, \sqrt{6m}\} \tag{11}$$

For each random tuple $(x, y, z) \in S$, define:

$$\mathcal{X}_{(x,y,z)} \quad = \quad \frac{f(x, y, z)}{D^+_{max}}. \tag{12}$$

Note that $\mathcal{X}_{(x,y,z)}$ is a random variable between 0 and 1 because $f(x, y, z) \le D^+(x)$, while $D^+(x)$ is at most $(1 + \epsilon/2) \cdot d^+(x)$ (see (5)), which in turn is at most D^+_{max} by Lemma 13. Set

$$\mathcal{X} \quad = \quad \sum_{(x,y,z) \in S} \mathcal{X}_{(x,y,z)} = \frac{\Lambda}{D^+_{max}} \tag{13}$$

where the last equality used (7).

▶ **Lemma 14.** $\left(1 - \frac{\epsilon}{2}\right) \frac{s \cdot T}{m \cdot D^+_{max}} \le \mathbf{E}[\mathcal{X}] \le \left(1 + \frac{\epsilon}{2}\right) \frac{s \cdot T}{m \cdot D^+_{max}}$.

Proof. On condition that (x, y) equals edge (u, v) in G^+, the random variable $f(x, y, z)$ takes value $D^+(u)$ if $(v, z) \in E^+$ or 0 otherwise. $(v, z) \in E^+$ if and only if z is a common out-neighbor of u and v. Hence:

$$\begin{aligned}
\mathbf{E}[f(x,y,z)] &= \frac{1}{m} \sum_{(u,v)\in E^+} \frac{|N^+(u) \cap N^+(v)|}{d^+(u)} \cdot D^+(u) \\
\text{(by (5))} \quad &\leq \frac{1}{m} \sum_{(u,v)\in E^+} |N^+(u) \cap N^+(v)| \cdot (1+\epsilon/2) = (1+\epsilon/2) \cdot \frac{T}{m}.
\end{aligned}$$

It thus follows from (12) and (13) that $\mathbf{E}[\mathcal{X}] \leq (1+\epsilon/2)\frac{s \cdot T}{m \cdot D^+_{max}}$.

Analogously, applying the fact that $D^+(u)/d^+(u) \geq 1 - \epsilon/2$ for all $u \in V$ leads to $\mathbf{E}[\mathcal{X}] \geq (1-\epsilon/2)\frac{s \cdot T}{m \cdot D^+_{max}}$. ◀

We will proceed differently from here, depending on the comparison between T and $\Gamma(m)$.

B.1 When $T \geq \Gamma(m)$

We will prove that $s = \tilde{O}((1/\epsilon)^2 \cdot M^{1.5}/\Gamma(M))$ ensures:

$$\mathbf{Pr}\left[\left|\Lambda \cdot \frac{m}{s} - T\right| \geq \epsilon \cdot T\right] \leq \frac{1}{m^3} \tag{14}$$

▶ **Lemma 15.** *We can choose an $s = \tilde{O}((1/\epsilon)^2 \cdot M^{1.5}/\Gamma(M))$ to guarantee*

$$\mathbf{Pr}\left[|\mathcal{X} - \mathbf{E}[\mathcal{X}]| \geq \frac{\epsilon}{2} \cdot \frac{s \cdot T}{m \cdot D^+_{max}}\right] \leq \frac{1}{m^3}.$$

Proof.

$$\begin{aligned}
&\mathbf{Pr}\left[|\mathcal{X} - \mathbf{E}[\mathcal{X}]| \geq \frac{\epsilon}{2} \cdot \frac{s \cdot T}{m \cdot D^+_{max}}\right] \\
\text{(by Lemma 14)} \quad \leq\ &\mathbf{Pr}\left[|\mathcal{X} - \mathbf{E}[\mathcal{X}]| \geq \frac{\epsilon}{2} \cdot \frac{\mathbf{E}[X]}{1+\epsilon/2}\right] \leq \mathbf{Pr}\left[|\mathcal{X} - \mathbf{E}[\mathcal{X}]| \geq \frac{\epsilon}{3} \cdot \mathbf{E}[X]\right] \\
\text{(by (9))} \quad \leq\ &2\exp\left(-\left(\frac{\epsilon}{3}\right)^2 \frac{\mathbf{E}[\mathcal{X}]}{3}\right) \\
\text{(by Lemma 14)} \quad \leq\ &2\exp\left(-\left(\frac{\epsilon}{3}\right)^2 \frac{s \cdot T}{3m \cdot D^+_{max}} \cdot (1-\epsilon/2)\right) \\
\leq\ &2\exp\left(-\left(\frac{\epsilon}{3}\right)^2 \frac{s \cdot T}{6m \cdot D^+_{max}}\right)
\end{aligned}$$

which is at most $1/m^3$ for $s = O(\frac{mD^+_{max}}{\epsilon^2 \cdot T} \log m)$. The claim follows from $D^+_{max} = O(\sqrt{m})$ (see (11)), $T \geq \Gamma(m) \geq \Gamma(M)$, and $m \leq 2M$. ◀

The lemma implies (14) because

$$\begin{aligned}
&\mathbf{Pr}\left[|\mathcal{X} - \mathbf{E}[\mathcal{X}]| \geq \frac{\epsilon}{2} \cdot \frac{s \cdot T}{m \cdot D^+_{max}}\right] \\
=\ &\mathbf{Pr}\left[\mathcal{X} \geq \mathbf{E}[\mathcal{X}] + \frac{\epsilon}{2} \cdot \frac{s \cdot T}{m \cdot D^+_{max}} \text{ or } \mathcal{X} \leq \mathbf{E}[\mathcal{X}] - \frac{\epsilon}{2} \cdot \frac{s \cdot T}{m \cdot D^+_{max}}\right] \\
\text{(by Lemma 14)} \quad \geq\ &\mathbf{Pr}\left[\mathcal{X} \geq (1+\epsilon)\frac{s \cdot T}{m \cdot D^+_{max}} \text{ or } \mathcal{X} \leq (1-\epsilon)\frac{s \cdot T}{m \cdot D^+_{max}}\right] \\
\text{(by (13))} \quad =\ &\mathbf{Pr}\left[\Lambda \cdot \frac{m}{s} \geq (1+\epsilon)T \text{ or } \Lambda \cdot \frac{m}{s} \leq (1-\epsilon)T\right] \\
=\ &\mathbf{Pr}\left[\left|\Lambda \cdot \frac{m}{s} - T\right| \geq \epsilon \cdot T\right].
\end{aligned}$$

B.2 When $0 < T < \Gamma(m)$

We will prove that $s = \tilde{O}((1/\epsilon)^2 \cdot M^{1.5}/\Gamma(M))$ ensures:

$$\mathbf{Pr}\left[\left|\Lambda \cdot \frac{m}{s} - T\right| \geq \epsilon \cdot \Gamma(m)\right] \quad \leq \quad \frac{1}{m^3} \tag{15}$$

Since there is at least one triangle, $\mathcal{X}_{x,y,z}$ (see (12)) has expectation strictly greater than 0. By (13), this means $\mathbf{E}[\mathcal{X}] > 0$.

▶ **Lemma 16.** *We can choose an $s = \tilde{O}((1/\epsilon)^2 \cdot M^{1.5}/\Gamma(M))$ to guarantee*

$$\mathbf{Pr}\left[|\mathcal{X} - \mathbf{E}[\mathcal{X}]| \geq \frac{\epsilon}{2} \cdot \frac{s \cdot \Gamma(m)}{m \cdot D^+_{max}}\right] \quad \leq \quad \frac{1}{m^3}.$$

Proof.

$$\mathbf{Pr}\left[|\mathcal{X} - \mathbf{E}[\mathcal{X}]| \geq \frac{\epsilon}{2} \cdot \frac{s \cdot \Gamma(m)}{m \cdot D^+_{max}}\right] = \mathbf{Pr}\left[|\mathcal{X} - \mathbf{E}[\mathcal{X}]| \geq \frac{\epsilon}{2} \cdot \frac{s \cdot \Gamma(m)}{m \cdot D^+_{max} \cdot \mathbf{E}[\mathcal{X}]} \cdot \mathbf{E}[\mathcal{X}]\right] \tag{16}$$

Setting $\gamma = \frac{\epsilon \cdot s \cdot \Gamma(m)}{2m \cdot D^+_{max} \cdot \mathbf{E}[\mathcal{X}]}$, we distinguish two cases.

Case 1: $\gamma \leq 1$. By (9), we have

$$\begin{aligned}
(16) \quad &\leq \quad 2\exp\left(-\gamma^2 \cdot \frac{\mathbf{E}[\mathcal{X}]}{3}\right) = 2\exp\left(-\left(\frac{\epsilon \cdot s \cdot \Gamma(m)}{2m \cdot D^+_{max}}\right)^2 \cdot \frac{1}{3\,\mathbf{E}[\mathcal{X}]}\right) \\
\text{(by Lemma 14)} \quad &\leq \quad 2\exp\left(-\left(\frac{\epsilon \cdot s \cdot \Gamma(m)}{2m \cdot D^+_{max}}\right)^2 \cdot \frac{1}{3(1+\epsilon/2)\frac{s \cdot T}{m \cdot D^+_{max}}}\right) \\
&\leq \quad 2\exp\left(-\frac{\epsilon^2 s \cdot (\Gamma(m))^2}{18m \cdot T \cdot D^+_{max}}\right) \\
\text{(by } \Gamma(m) > T) \quad &\leq \quad 2\exp\left(-\frac{\epsilon^2 s \cdot \Gamma(m)}{18m \cdot D^+_{max}}\right)
\end{aligned}$$

which is at most $1/m^3$ for $s = O(\frac{mD^+_{max} \cdot \log m}{\epsilon^2 \Gamma(m)})$. The claim follows from $D^+_{max} = O(\sqrt{m})$, $\Gamma(m) \geq \Gamma(M)$, and $m \leq 2M$.

Case 2: $\gamma > 1$. Since $\mathcal{X} \geq 0$, we have

$$\begin{aligned}
(16) \quad &= \quad \mathbf{Pr}\Big[\mathcal{X} - \mathbf{E}[\mathcal{X}] \geq \gamma \cdot \mathbf{E}[\mathcal{X}]\Big] \\
\text{(by (10))} \quad &\leq \quad \exp\left(-\frac{1+\gamma}{6}\,\mathbf{E}[\mathcal{X}]\right) \leq \exp\left(-\frac{\gamma}{6}\,\mathbf{E}[\mathcal{X}]\right) \\
&\leq \quad \exp\left(-\frac{\epsilon \cdot s \cdot \Gamma(m)}{12m \cdot D^+_{max} \cdot \mathbf{E}[\mathcal{X}]} \cdot \mathbf{E}[\mathcal{X}]\right) = \exp\left(-\frac{\epsilon \cdot s \cdot \Gamma(m)}{12m \cdot D^+_{max}}\right)
\end{aligned}$$

which is at most $1/m^3$ for $s = O(\frac{mD^+_{max} \cdot \log m}{\epsilon \cdot \Gamma(m)})$. The claim follows from $D^+_{max} = O(\sqrt{m})$, $\Gamma(m) \geq \Gamma(M)$, and $m \leq 2M$. ◀

The lemma implies (15) because

$$
\begin{aligned}
& \mathbf{Pr}\left[\left|\mathcal{X} - \mathbf{E}[\mathcal{X}]\right| \geq \frac{\epsilon}{2} \cdot \frac{s \cdot \Gamma(m)}{m \cdot D^+_{max}}\right] \\
= \; & \mathbf{Pr}\left[\mathcal{X} \geq \mathbf{E}[\mathcal{X}] + \frac{\epsilon}{2} \cdot \frac{s \cdot \Gamma(m)}{m \cdot D^+_{max}} \text{ or } \mathcal{X} \leq \mathbf{E}[\mathcal{X}] - \frac{\epsilon}{2} \cdot \frac{s \cdot \Gamma(m)}{m \cdot D^+_{max}}\right] \\
\text{(by Lemma 14)} \geq \; & \mathbf{Pr}\Big[\mathcal{X} \geq \left(1 + \frac{\epsilon}{2}\right) \frac{s \cdot T}{m D^+_{max}} + \frac{\epsilon s \cdot \Gamma(m)}{2 m D^+_{max}} \text{ or} \\
& \mathcal{X} \leq \left(1 - \frac{\epsilon}{2}\right) \frac{s \cdot T}{m D^+_{max}} - \frac{\epsilon s \cdot \Gamma(m)}{2 m D^+_{max}}\Big] \\
\text{(by } T < \Gamma(m)) \geq \; & \mathbf{Pr}\left[\mathcal{X} \geq \frac{s \cdot T}{m \cdot D^+_{max}} + \frac{\epsilon s \cdot \Gamma(m)}{m \cdot D^+_{max}} \text{ or } \mathcal{X} \leq \frac{s \cdot T}{m \cdot D^+_{max}} - \frac{\epsilon s \cdot \Gamma(m)}{m \cdot D^+_{max}}\right] \\
\text{(by (13))} = \; & \mathbf{Pr}\left[\Lambda \cdot \frac{m}{s} \geq T + \epsilon \cdot \Gamma(m) \text{ or } \Lambda \cdot \frac{m}{s} \leq T - \epsilon \cdot \Gamma(m)\right] \\
= \; & \mathbf{Pr}\left[\left|\Lambda \cdot \frac{m}{s} - T\right| \geq \epsilon \cdot \Gamma(m)\right].
\end{aligned}
$$

B.3 When $T = 0$

In this case, every random tuple must have contribution 0 (see (6)). Thus, Λ must be 0, and hence, so is our estimate.

C Deletion Algorithm of the DATC Structure

C.1 Deletion: When D Will Still Be Good

Suppose that we are deleting an edge $\{u^*, v^*\}$ in G. This section discusses the scenario where D is still good after $d(u^*)$ and $d(v^*)$ decrease by 1. Assume, without loss of generality, that $\{u^*, v^*\}$ points from u^* to v^* in G^+.

Every $(x, y, z) \in S$ with $x = u^*$ and $y = v^*$ should be replaced with a new random tuple. For this purpose, we remove the entire $S_{(u^*, v^*, \perp)}$ from S, regenerate the same the same number of random tuples, and add them to S. By Lemma 7, this can be done in $\tilde{O}(s/m)$ time w.h.p.

Now consider a tuple $(x, y, z) \in S$ with $x \neq u^*$ or $y \neq v^*$. After the deletion, (x, y) remains as a uniformly random edge in from E^+. Nevertheless, we still need to make sure that z is a random out-neighbor of x, and that Λ is correct:

- Case 1: $x = u^*$ and $z = v^*$. We remove (x, y, z) from S, select an out-neighbor w of u^* uniformly at random, and add (u^*, y, w) to S.
- Case 2: $y = u^*$ and $z = v^*$. As the deleted edge (u^*, v^*) breaks the triangle formed by x, u^*, and v^*, Λ should be decreased by $D^+(x)$.

Regarding implementation, all the tuples of Case 1 can be found in $\tilde{O}(|S_{(u^*, \perp, v^*)})|$ time, which is $\tilde{O}(s/m)$ w.h.p. by Lemma 7; this is also the time spent on Case 1 in total. For Case 2, the overall amount of reduction on Λ (summing up over all tuples of Case 2) is simply Ξ_{u^*, v^*}, which can be retrieved in $\tilde{O}(1)$ time; and then Λ can be adjusted in constant time.

Finally, if $D^+(u^*)$ no longer satisfies (5), we simply call **`fix-Dplus`**(u^*) (Section 3.3).

In summary, the deletion runs in $\tilde{O}(s/m)$ time w.h.p, plus the cost of at most one call to **`fix-Dplus`**.

C.2 Deletion: When D Will Go Bad

We now consider the scenario where D violates (4) after $\{u^*, v^*\}$ is deleted. Similar to Section 3.4, we reduce the case to Section C.1 by first modifying D such that it will still be good after the deletion. Due to symmetry, it suffices to discuss only the situation where $D(u^*)$ needs to be fixed.

The fix is performed by **`fix-D-del`**(u), which has the constraint:

Invariant: when called:
- D is good
- $d(u) < D(u)$ and
- $D(u) - d(u) = \Omega(D(u))$.

At the end of **`fix-D-del`**(u), $D(u) = d(u)$. It is rudimentary to verify that D will still be good after $d(u)$ drops by 1.

Rationale behind `fix-D-del`(u). We decrease $D(u)$ to $d(u)$, which may affect the direction of an edge in G^+ incident on u: if the direction was (v, u) before, it may now be flipped to (u, v).

Fix a tuple $(x, y, z) \in S$. Consider an arbitrary edge (v, u) that has been flipped to (u, v). The next discussion clarifies all the cases that require modifications:
- Case 1: $x \neq v$ and $x \neq u$. (x, y, z) still remains as a random tuple, but its contribution may change:
  - Case 1.1: $y = u$ and $z = v$. $f(x, y, z)$ will grow from 0 to $D^+(x)$. Accordingly, Λ needs to be increased by $D^+(x)$.
  - Case 1.2: $y = v$ and $z = u$. $f(x, y, z)$ will drop from $D^+(x)$ to 0. Accordingly, Λ needs to be decreased by $D^+(x)$.
- Case 2: $x = v$ and $y = u$. The tuple (v, u, z) should be replaced by (u, v, w) where w is a (uniformly) random out-neighbor of u.
- Case 3: $x = v$, $y \neq u$, and $z = u$. The tuple (v, y, u) should be replaced by (v, y, w) where w is a (uniformly) random out-neighbor of v.
- Case 4: $x = u$ (which implies $y \neq v$ and $z \neq v$). z should be replaced by v with probability $1/d^+(u)$. If the replacement occurs, the tuple (u, y, z) is said to be *outneighbor-replaced*.

Note the similarity to the cases in Section 3.4.

Algorithm `fix-D-del`(u). Set $D(u) = d(u)$ and flip the edges of u in G^+ wherever needed.

Given each neighbor v of u such that $\{u, v\}$ was flipped, we
- (for Case 1) retrieve $\Xi_{u,v}$ and $\Xi_{v,u}$ (from the auxiliary structures), and increase Λ by $\Xi_{u,v} - \Xi_{v,u}$.
- (for Case 2) retrieve $S_{v,u,\perp}$; and then for each $(v, u, z) \in S_{v,u,\perp}$, delete (v, u, z) from S, pick an out-neighbor w of u uniformly at random, and add (u, v, w) to S.
- (for Case 3) retrieve $S_{v,\perp,u}$; and then for each $(v, y, u) \in S_{v,\perp,u}$ with $y \neq u$, delete (v, y, u) from S, pick an out-neighbor w of v uniformly at random, and add (v, y, w) to S.

Case 1 obviously takes $\tilde{O}(d(u) \cdot s/m))$ time w.h.p. By Lemma 7, Cases 2 and 3 can also be handled in the same cost.

Next, we attend to Case 4. Consider any neighbor v of u with $\{u, v\}$ flipped. The number k_u of outneighbor-replaced tuples (x, y, z) with $x = u$ follows the binomial distribution $B(|S_{u,\perp,\perp}|, 1/d^+(u))$. This, together with Lemma 7, shows that $k_u = \tilde{O}(d^+(u) \cdot \frac{s}{m} \cdot \frac{1}{d^+(u)}) = \tilde{O}(s/m)$ w.h.p. We draw a WoR sample set of size k_u from $|S_{u,\perp,\perp}|$ in $\tilde{O}(k_u) = \tilde{O}(s/m)$ time. Every tuple (u, y, z) drawn is modified to (u, y, v) in $\tilde{O}(1)$ time. The total cost of Case 4 is $\tilde{O}(d(u) \cdot s/m)$ w.h.p.

Finally, if D^+ is bad, we remedy it in the same way as in Section 3.4.

In summary, **`fix-D-del`**(u) runs in $\tilde{O}(d(u) \cdot s/m)$ time w.h.p., plus the cost of all the calls to **`fix-Dplus`** at the end. The invariant ensures that $d(u) < \mathcal{D}_{old}$ where $\mathcal{D}_{old}$ is the value of $D(u)$ at the beginning of **`fix-D-del`**(u).

C.3 Analysis

The analysis is a straightforward adaptation of the argument in Section 3.5. It suffices to point out some key changes:

- The invariant of **fix-D-del** makes sure that $\Omega(\mathcal{D}_{old})$ edges incident on u have been removed since the last call to **fix-D-del**(u), where $\mathcal{D}_{old}$ is the value of $D(u)$ at the beginning of **fix-D-del**(u).
- When an edge (u^*, v^*) is deleted from G, we give u^* a token.
- During the execution of **fix-D-del**(u), when we flip an in/out-edge of u with respect an its in/out-neighbor v, we give a token to both u and v.
- Lemma 10 should be replaced with: if the total number of edge insertions/deletions is n_{upd}, the number of tokens generated is $O(n_{upd})$.

D Deletion Algorithm of the DETC Structure

Update $\boldsymbol{T}$. Suppose that we are deleting (u, v) from G^+. The possible cases for a triangle involving u and v are the same as in Figure 3. The number of such triangles can be found in the same manner as in the insertion algorithm using $\tilde{O}(\alpha + \lambda)$ time. After that, T is updated in constant time.

Update $\boldsymbol{\mathcal{I}_A}$ and $\boldsymbol{A}$. If v is heavy, for every heavy out-neighbor $w \neq v$ of u, we decrease $I_{\{v,w\}}$ by 1. If $I_{\{v,w\}} = 0$, $\{v, w\}$ is removed from A. The time is $\tilde{O}(d^+(u)) = \tilde{O}(\alpha)$.

Vertex u (the case of v is similar) may have just turned from heavy to light. We examine every in-neighbor x of u in G. For each heavy out-neighbor y of x, remove $\{u, y\}$ from A. This takes $\tilde{O}(\alpha\lambda)$ time in total. We charge the time on the $\Omega(\lambda)$ edges of u that have been removed since u turned heavy last time. After amortization, the deletion of each of those edges bears only $\tilde{O}(\alpha)$ time.

We conclude that the deletion time is $\tilde{O}(\alpha + \lambda)$ amortized.

E Proof of Lemma 5

In [21], Henzinger, Krinninger, Nanongkai, and Saranurak defined the *online vector-matrix-vector multiplication* problem, which they abbreviated as the OuMv problem. An algorithm is allowed to pre-process an $n \times n$ matrix $\boldsymbol{M}$ in $\mathrm{poly}(n)$ time. Then, given n pairs of vectors $(\boldsymbol{u}_i, \boldsymbol{v}_i)$ where $\boldsymbol{u}_i$ is a $1 \times n$ vector and $\boldsymbol{v}_i$ is an $n \times 1$ vector, the algorithm is required to compute $\boldsymbol{u}_i\boldsymbol{M}\boldsymbol{v}_i$. Only after $\boldsymbol{u}_i\boldsymbol{M}\boldsymbol{v}_i$ has been output will $(\boldsymbol{u}_{i+1}, \boldsymbol{v}_{i+1})$ be given (for $i \in [n-1]$). Every element in $\boldsymbol{M}$, and in each $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$ is either 0 or 1; and addition and multiplication are performed as OR and AND, respectively. The *cost* of an algorithm is the total time spent on the n pairs of vectors. The following was proved in [21]:

▶ **Lemma 17** ([21]). *Subject to the OMv-conjecture, no algorithm can solve the OuMv problem with probability at least 2/3 in $O(n^{3-\delta})$ time for any constant $\delta > 0$.*

We will prove Lemma 5 by reducing the OuMv problem to DTS. Suppose that an algorithm $\mathcal{A}$ is able to maintain a DTD structure capable of performing an update in $O(m^{0.5-\delta'})$ expected amortized time and a query in $O(m^{1-\delta'})$ time, for some $\delta' > 0$. We will leverage $\mathcal{A}$ to obtain an algorithm that contradicts Lemma 17.

It suffices to consider that $\boldsymbol{M}$ has at least n 1's. Otherwise, $\boldsymbol{u}\boldsymbol{M}\boldsymbol{v}$ can be easily calculated in $O(n)$ time for any $1 \times n$ vector $\boldsymbol{u}$ and $n \times 1$ vector $\boldsymbol{v}$. In this case, the OuMv problem can be settled in $O(n^2)$ time.

In the preprocessing stage (of OuMv), we create a graph G as follows:

- G has a vertex corresponding to each row in $\boldsymbol{M}$, and a vertex corresponding to each column in $\boldsymbol{M}$. In addition, there is an extra vertex denoted as ψ. The total number of vertices in $2n+1$.
- For each cell $\boldsymbol{M}[i,j]=1$ $(i,j\in[n])$, G has an edge connecting the vertex of row i with the vertex of column j. The number m of edges satisfies $n\le m=O(n^2)$.

We construct a DTD structure on G using $\mathcal{A}$. The time required is obviously $\mathrm{poly}(n)$.

We process an incoming vector pair $(\boldsymbol{u},\boldsymbol{v})$ of the OuMV problem as follows:

1. For each $i\in[n]$ such that $\boldsymbol{u}[i]=1$, add an edge between ψ and the vertex corresponding to row i. For each $j\in[n]$ such that $\boldsymbol{v}[j]=1$, add an edge between ψ and the vertex corresponding to column j.
2. Issue a DTD query to detect whether G has a triangle.
3. Remove all the edges added in Step 1.

It was proved in [21] (Lemma 3.3 therein) that $\boldsymbol{uMv}=1$ if and only if the query in Step 2 reports "yes".

The number m of edges satisfies $n\le m=O(n^2)$ at all times. The 3 steps require at most $2n$ update operations and 1 query on the DTD structure, which (by our assumption on $\mathcal{A}$) finish in $O(n\cdot m^{0.5-\delta'}+m^{1-\delta'})=O(n^{2-2\delta'})$ expected time.

After processing n vector pairs, with probability at least $1-\frac{n}{m^2}\ge 1-\frac{n}{n^2}=1-1/n$, all the n DTD queries issued are correct. We thus have obtained an algorithm solving the OuMv-problem with probability at least $1-1/n$ in $O(n^{3-2\delta'})$ expected time. By Markov's inequality, with probability at least $3/4$, the actual running time is at most 4 time higher. Therefore, our algorithm solves the OuMv-problem in $O(n^{3-2\delta'})$ time with probability at least $1-(1/n+1/4)$ which is greater than $2/3$ for $n>12$. This contradicts Lemma 17.

References

1. Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.
2. Sepehr Assadi, Michael Kapralov, and Sanjeev Khanna. A simple sublinear-time algorithm for counting arbitrary subgraphs via edge sampling. In *Innovations in Theoretical Computer Science (ITCS)*, pages 6:1–6:20, 2019.
3. Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 623–632, 2002.
4. Suman K. Bera and Amit Chakrabarti. Towards tighter space bounds for counting triangles and other substructures in graph streams. In *Proceedings of Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 11:1–11:14, 2017.
5. Suman K. Bera and C. Seshadhri. How the degeneracy helps for triangle counting in graph streams. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 457–467, 2020.
6. Edvin Berglin and Gerth Stølting Brodal. A simple greedy algorithm for dynamic graph orientation. *Algorithmica*, 82(2):245–259, 2020.
7. Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 303–318, 2017.
8. Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. *ACM Transactions on Database Systems (TODS)*, 43(2):7:1–7:32, 2018.
9. Vladimir Braverman, Rafail Ostrovsky, and Dan Vilenchik. How hard is counting triangles in the streaming model? In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 244–254, 2013.
10. Laurent Bulteau, Vincent Froese, Konstantin Kutzkov, and Rasmus Pagh. Triangle counting in dynamic graph streams. *Algorithmica*, 76(1):259–278, 2016.

11 Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 253–262, 2006.

12 Yu Chen and Ke Yi. Random sampling and size estimation over cyclic joins. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 7:1–7:18, 2020.

13 N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal of Computing*, 14(1):210–223, 1985.

14 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

15 Graham Cormode and Hossein Jowhari. A second look at counting triangles in graph streams (corrected). *Theoretical Computer Science*, 683:22–30, 2017.

16 Talya Eden, Amit Levi, Dana Ron, and C. Seshadhri. Approximately counting triangles in sublinear time. *SIAM Journal on Computing*, 46(5):1603–1646, 2017.

17 David Eppstein and Emma S. Spiro. The h-index of a graph and its application to dynamic subgraph statistics. *J. Graph Algorithms Appl.*, 16(2):543–567, 2012.

18 David García-Soriano and Konstantin Kutzkov. Triangle counting in streamed graphs via small vertex covers. In *SIAM International Conference on Data Mining (SDM)*, pages 352–360, 2014.

19 Isaac Goldstein, Moshe Lewenstein, and Ely Porat. On the hardness of set disjointness and set intersection with bounded universe. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 7:1–7:22, 2019.

20 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 21–30, 2015.

21 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. *CoRR*, abs/1511.06773, 2015.

22 Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. I/O-Efficient Algorithms on Triangle Listing and Counting. *ACM Transactions on Database Systems (TODS)*, 39(4):27:1–27:30, 2014.

23 Madhav Jha, C. Seshadhri, and Ali Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *Proceedings of ACM Knowledge Discovery and Data Mining (SIGKDD)*, pages 589–597, 2013.

24 Hossein Jowhari and Mohammad Ghodsi. New streaming algorithms for counting triangles in graphs. In *Computing and Combinatorics*, pages 710–716, 2005.

25 John Kallaugher and Eric Price. A hybrid sampling scheme for triangle counting. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1778–1797, 2017.

26 Daniel M. Kane, Kurt Mehlhorn, Thomas Sauerwald, and He Sun. Counting arbitrary subgraphs in data streams. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 598–609, 2012.

27 Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting triangles under updates in worst-case optimal time. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 4:1–4:18, 2019.

28 Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Dynamic set intersection. *CoRR*, abs/1407.6755, 2014.

29 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1272–1287, 2016.

30 Madhusudan Manjunath, Kurt Mehlhorn, Konstantinos Panagiotou, and He Sun. Approximate counting of cycles in streams. In *Proceedings of European Symposium on Algorithms (ESA)*, 2011.

31 Andrew McGregor, Sofya Vorotnikova, and Hoa T. Vu. Better algorithms for counting triangles in data streams. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 401–411, 2016.

32 Rasmus Pagh and Charalampos E. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters (IPL)*, 112(7):277–281, 2012.
33 Mihai Patrascu. Towards polynomial lower bounds for dynamic problems. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 603–610, 2010.
34 A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and sampling triangles from a graph stream. *Proceedings of the VLDB Endowment (PVLDB)*, 6(14):1870–1881, 2013.
35 C. Seshadhri, Ali Pinar, and Tamara G. Kolda. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining*, 7(4):294–307, 2014.
36 Cheng Sheng, Yufei Tao, and Jianzhong Li. Exact and approximate algorithms for the most connected vertex problem. *ACM Transactions on Database Systems (TODS)*, 37(2):12:1–12:39, 2012.
37 Charalampos E. Tsourakakis, Mihail N. Kolountzakis, and Gary L. Miller. Triangle sparsifiers. *J. Graph Algorithms Appl.*, 15(6):703–726, 2011.
38 Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.