Exact Dynamic Triangle Counting
Reading Group Algorithms Companion Paper

Daniel Schmitt

February 14, 2022

Abstract

In this paper, we consider the problem of counting and enumerating all triangles in a graph. A recent result by Kara et al. [8] maintains the number of triangles under edge insertions and deletions and allows enumeration of all triangles with constant delay. Their algorithm was formulated for relational database systems using concepts of incremental view maintenance and fractional databases to perform the updates. We simplify and formulate their algorithm using typical terms and concepts of graph theory.

1 Introduction

Graphs are often the prefered representation for technological, biological, and sociological systems. Web graphs are used to represent websites and hyperlinks between them. Social graphs represent users and their interactions. Such graphs are often analyzed using the local clustering coefficient or the transitivity ratio that both require counting triangles in the graph. Other applications of triangles include community detection, spam detection, and recommender systems. For a larger list of applications of triangles in graphs, we refer to Tsourakakis et al. [14].

In addition to the exact, dynamic, and sequential setting that we consider in this paper, other settings have also been broadly studied. The fastest static algorithm for exact triangle counting is the method by Alon et al. [2] that runs in $\mathcal{O}(m^{\frac{2\omega}{\omega+1}})$ time, where ω is the exponent of fast matrix multiplication ($\omega < 2.3729$ [1]). The approximate triangle counting problem has been extensively studied (e.g., [4, 11]) in the *streaming model*, where one or a constant number of passes over E are allowed. In *dynamic graph streams*, the streams can include both edge insertions and deletions [3].

In this paper, we consider the recent algorithm by Kara et al. [8] that was formulated for relational database systems. Inside the database, we store three relations $R[a, b]$, $S[b, c]$, $T[c, a]$ and want to count or enumerate all a, b, c such that the tuples $(a, b) \in R$, $(b, c) \in S$, and $(c, a) \in T$. This can be regarded as a tripartite graph with edge sets R, S, and T, where each relation connects nodes of two different partitions. The authors show that both the total number of triangles Δ_0 and the possibility of enumerating all triangles Δ_3 in the graph can be maintained under edge insertions and deletions in sublinear time $\mathcal{O}(m^{\frac{1}{2}})$, where m is the number of edges. It is easy to see that the problem of triangle counting and enumeration for tripartite graphs is the same as in general graphs.

Lemma 1. The problems of finding and enumerating all triangles in general graphs and in tripartite graphs are identical.

Proof. It is obvious that any algorithm solving the problems in a general graph can also solve the problems in tripartite graphs. For the other direction, we use a common trick in triangle counting. For a given general undirected graph $G = (V, E)$, we from a new graph G' by creating three copies of its nodes V_1, V_2, V_3 and a new edge set E'. Now, to avoid overcounting triangles, we "orient" the undirected edges (u, v) using any order $<$ on the nodes. For each $(u, v) \in E$, we only add edges (u_1, v_2), (u_1, v_3), and (u_2, v_3) between the copies $u_1 \in V_1$, $u_2 \in V_2$, $v_2 \in V_2$, $v_3 \in V_3$, if $u < v$.

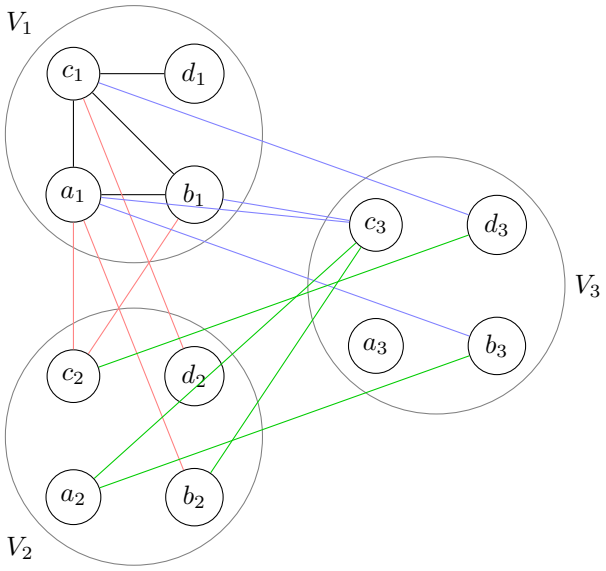


Figure 1: Reduction of a graph to a tripartite graph using the order $a < b < c < d$. The original edges are drawn in black. The new edges between V_1 and V_2, V_1 and V_3, and V_2 and V_3 are drawn in red, blue, and green, respectively.

Obviously, G' is tripartite. For any triangle in G, we can order its nodes from smallest to largest (u, v, w). Therefore, we can find the triangle (u_1, v_2, w_3) in G'. On the other hand, every triangle in G' has to be of the form (u_1, v_2, w_3) with $u_1 < v_2 < w_3$, thereby corresponding to exactly one triangle in G. An example is shown in Figure 1. □

This allows us to use the main result from Kara et al. [8] in general graphs.

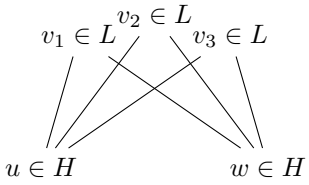
Theorem 1 (Kara et al. [8])**.** There exists an algorithm that can preprocess a graph $G = (V, E)$ with $n = |V|$ and $m = |E|$ in $\mathcal{O}(m^{\frac{3}{2}})$ time, uses $\mathcal{O}(m^{\frac{3}{2}})$ space, allows insertions and deletions to G in amortized $\mathcal{O}(m^{\frac{1}{2}})$ time, and allows finding the number of triangles in constant time or enumerating all triangles in time proportional to the number of triangles in G.

2 Dynamic Triangle Counting in General Graphs

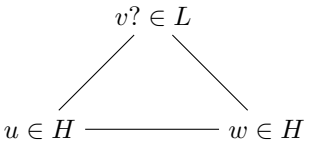
As we have seen, the algorithm by Kara et al. [8] that was formulated for relational database systems can be used to find and count all triangles in general graphs. In this section, we adapt their algorithm to work in general graphs without any reduction. We reuse their idea of ϵ-partitioning the nodes into sets of nodes with low and high degree.

Definition 1 (ϵ-Partition)**.** Let $\epsilon \in [0, 1]$. The nodes of an undirected graph $G = (V, E)$ with $n = |V|$ and $m = |E|$ are ϵ-partitioned into a heavy part H and a light part L, if $\frac{\hat{m}}{4} \leq m < \hat{m}$, all nodes $v \in H$ have degree $deg(v) \geq \frac{3}{2}\hat{m}^{\epsilon}$, all nodes $v \in L$ have degree $deg(v) < \frac{1}{2}\hat{m}^{\epsilon}$, $H \cap L = \emptyset$, and $H \cup L = V$. This partiton can be constructed in $O(n)$ time.

$\hat{m}$ is used as an estimation of m to avoid changing the degree threshold after every edge update that changes m. As already seen in Theorem 1, setting $\epsilon = \frac{1}{2}$ minimizes the complexity. For simplicity, we will only consider this case. Since the sum over all node degrees is $2m$, the size of H is bounded from above by $\frac{4m}{3\hat{m}^{\frac{1}{2}}} = \mathcal{O}(m^{\frac{1}{2}})$. Also, every node in L has at most $\frac{1}{2}m^{\epsilon} = \mathcal{O}(\hat{m}^{\frac{1}{2}})$ neighbors. We partition the edges $(u, v) \in E$ into the sets HH, HL, LH, LL, where u and v are of



(a) P_2 contains all paths of length 2. For Δ_0, we only store the number of paths from u to v as the label of (u, v) in P_2. For Δ_3, we store a single edge (u, v) multiple times with different labels $v_1, v_2, \ldots$.



(b) C_3 contains all edges that are part of a cycle of length 3, i.e., a triangle. (u, w) is stored in C_3, as there is a v that closes the cycle, but the v is unknown in C_3.

Figure 2: Auxiliary datastructures used for Δ_0 and Δ_3

high degree, u is of high and v of low degree, u is of low and v is of high degree, and both u and v are of low degree, respectively.[1]

We assume that the following operations can be performed on the graph datastructure and the edge partitions HH, HL, LH, LL:

1. Check whether a given node u is in H or in L in $\mathcal{O}(1)$.
2. For two given nodes u, v, check whether u and v are adjacent in $\mathcal{O}(1)$.
3. Enumerate all neighbors of a given node u in $\mathcal{O}(deg(u))$ and find $deg(u)$ in $\mathcal{O}(1)$.
4. Find, insert, and delete edges in $\mathcal{O}(1)$.

The graph datastructure and the edge partitions can be implemented, e.g., using hash tables instead of adjacency lists to allow those operations in *expected* constant time. Otherwise, balanced search trees allow all operations to be performed in worst-case logarithmic time, as critiqued by Lu et al. [10].

Additionally, we will use an auxiliary datastructure P_2 that stores all paths of length 2 (u, v, w) with $u, w \in H, v \in L$. Figure 2a shows a visualization of P_2. These 2-paths are often called *wedges* and are also used in triangle approximation (e.g., [9, 13]). We will use this datastructure when inserting or deleting edges (u, w), where both u and w are heavy nodes. Depending on whether we only want to count the number of triangles (Δ_0) or enumerate them (Δ_3), P_2 either has to only store the number of paths from u to w or all v that are on the paths from u to w. In any case, we can think of P_2 as an edge-labeled (multi-)graph, where the edge labels are either positive numbers (Δ_0) or nodes (Δ_3) and allowing the same operations as described for the graph data structure including finding the edge labels. P_2 can be initialized in $\mathcal{O}(m^{\frac{3}{2}})$ time and space by iterating over all edges $(u, v) \in HL$ and for each v finding all edges $(v, w) \in LH$, as there are at most $\mathcal{O}(m^{\frac{1}{2}})$ neighbors for each v. For each (u, v, w), we can update the edge (u, w) in P_2 by increasing the label (Δ_0) or adding a new edge (u, w) with label v (Δ_3) in constant time.

2.1 Single Edge Updates for Δ_0

Triangle Count Updates We will first consider the triangle count Δ_0. We can initialize its value by using a static triangle counting algorithm [2, 15, 12] in $\mathcal{O}(m^{\frac{3}{2}})$. As adding nodes can not result in added or removed triangles, we only have to consider edge updates and deletions. For an insertion (u, v), we have to find all edges (v, w) and (w, u) that will form a new triangle. For a deletion, we have to find the same edges to remove the triangles. In both cases, we consider the following two cases to update Δ_0:

[1] HL and LH contain the same edges and are just used for simpler notation. They do not have to be duplicated.

1. At least one of u or v is in L, w.l.o.g. $u \in L$: Enumerate all neighbors w of u, where w is either in H or in L, and check whether $(w, v) \in E$. As u has at most $\mathcal{O}(m^{\frac{1}{2}})$ neighbors and the check can be performed in $\mathcal{O}(1)$, this case takes $\mathcal{O}(m^{\frac{1}{2}})$ time.

2. Both u and v are heavy: The update for $w \in H$ is simple. u can have at most $\mathcal{O}(m^{\frac{1}{2}})$ heavy neighbors w and we can check whether $(w, v) \in E$ in $\mathcal{O}(1)$ for the same runtime as in the first case. The update for $w \in L$ is harder and requires our precomputed P_2 datastructure. We search for the edge (u, v) in P_2 for the count of wedges (u, w, v) and update the triangle count Δ_0 in $\mathcal{O}(1)$.

Auxiliary Datastructure Updates After updating the triangle count Δ_0, we also have to update the graph and our auxiliary datastructure. We can insert/delete (u, v) into/from the correct edge partition HH, HL, LH, LL by looking up the degree of u and v and inserting/deleting in $\mathcal{O}(1)$. If u and v are from different degree partitions, we also update P_2. W.l.o.g. $u \in L, v \in H$: We iterate over all $\mathcal{O}(m^{\frac{1}{2}})$ neighbors w of u and insert, remove or update the label of edge (w, v) in P_2.

To summarize, we can perform a single edge update in $\mathcal{O}(m^{\frac{1}{2}})$ time.

2.2 Single Edge Updates for Δ_3

Now, we will consider enumerating all triangles. As there can be up to $\mathcal{O}(m^{\frac{3}{2}})$ triangles in the graph, we only consider the *enumeration delay*, i.e., the time for finding the first triangle and the time between each following triangle. We perform similarly as for Δ_0, but initialize the set of all triangles $\Delta_3^{\boxminus LL}$, where at least two nodes are light, and the set of all triangles Δ_3^{HHH}, where all nodes are heavy, instead of only the triangle count Δ_0 in $\mathcal{O}(m^{\frac{3}{2}})$ time and space using existing static triangle enumeration algorithms [15, 12]. We can perform the updates to both sets using the same strategies as for Δ_0 in $\mathcal{O}(m^{\frac{1}{2}})$ time. Instead of increasing or decreasing Δ_0, we insert or delete the triangle in $\Delta_3^{\boxminus LL}$ and Δ_3^{HHH}, depending on how many nodes are heavy and light.

On-The-Fly Enumeration for Δ_3^{HHL} The remaining triangles in Δ_3^{HHL} are of the form (u, v, w), where $u, w \in H, v \in L$. In addition to P_2, we now require an additional auxiliary datastructure C_3 visualized in Figure 2b that stores all edges (u, w), if $(u, w) \in HH$ and $(u, w) \in P_2$. In other words, if $(u, w) \in C_3$, there exists both a path of length 1 and a path of length 2 from u to w, i.e., there exists a triangle containing u and w. We can initialize C_3 after initializing P_2 by computing the intersection of HH and P_2. As $|HH| = \mathcal{O}(m)$, this can be achieved in linear time as every edge lookup is performed in constant time.

Finally, we can compute the triangles in Δ_3^{HHL} on-the-fly with constant enumeration delay using the following strategy: For each edge (u, v) in C_3, perform a lookup for (u, v) in P_2 and yield all triangles (u, w, v) for all labels w found for edges (u, v) in P_2. As the lookup requires $\mathcal{O}(1)$ time and only returns nodes that close the cycle, the delay between finding triangles is constant.

Auxiliary Datastructure Updates For each edge update (u, v), where u and v are from different degree partitions, w.l.o.g. $u \in L, v \in H$, we might have to insert (resp. delete) edges in P_2 and C_3. For updating P_2, we proceed as for Δ_0, iterate over the neighborhood of u of size $\mathcal{O}(m^{\frac{1}{2}})$, and insert (resp. delete) edges with label u in P_2 instead of changing the label to reflect the new number of paths. Afterwards, for each of the at most $\mathcal{O}(m^{\frac{1}{2}})$ updated edges (u, v) in P_2, we can perform a lookup in HH to also insert or delete (u, v) in C_3, if such an edge exists.

For each edge update (u, v), where both u and v are heavy, we only have to perform a lookup in P_2 to check if there exists a path from u to v of length 2 and insert or delete the edge (u, v) in C_3.

To summarize, we can perform a single edge update in $\mathcal{O}(m^{\frac{1}{2}})$ and enumerate all triangles in time only proportional to the number of triangles in the graph.

2.3 Performing Multiple Updates

Until now, we only discussed single edge updates. After multiple updates, it might be the case that our estimation $\hat{m}$ of the actual number of edges m might fall out of the range $\frac{\hat{m}}{4} \leq m < \hat{m}$. Additionally, edge updates can increase or decrease the degree of a node, which might change a heavy node to a light node or vice versa. In both cases, we have to perform significantly more work to either repartition V using a new threshold (*major rebalancing*) or delete a node from one partition and add it to the other (*minor rebalancing*). This superlinear time is, however, amortized over multiple sublinear updates, allowing for $\mathcal{O}(m^{\frac{1}{2}})$ amortized update time. As the proof by Kara et al. [8] can be used without significant changes, we only sketch its correctness.

Lemma 2. Major rebalancing requires $\mathcal{O}(m^{\frac{1}{2}})$ amortized update time.

Proof sketch. If $\hat{m}$ falls out of the range $\frac{\hat{m}}{4} \leq m < \hat{m}$, we reinitialize the $\epsilon - partition$ and perform initialization of all auxiliary datastructures in $\mathcal{O}(m^{\frac{3}{2}})$ time. As we need $\Omega(m)$ updates between each major rebalance, the amortized time complexity of major rebalancing is $\mathcal{O}(m^{\frac{1}{2}})$. □

Lemma 3. Minor rebalancing requires $\mathcal{O}(m^{\frac{1}{2}})$ amortized update time.

Proof sketch. If a heavy node u becomes a light node or vice versa, we have to delete and reinsert its adjacent edges in HH, HL, LH, LL and the dependent auxiliary datastructures P_2 and C_3. At the point in time when u changes its partition membership, it has degree $\mathcal{O}(m^{\frac{1}{2}})$. As we have seen for both Δ_0 and Δ_3, single edge updates take $\mathcal{O}(m^{\frac{1}{2}})$ time, i.e., minor rebalancing takes $\mathcal{O}(m)$ time. As minor rebalancing can only happen after $\Omega(m^{\frac{1}{2}})$ updates due to the "gap" in the threshold of light and heavy nodes, the amortized time complexity of minor rebalancing is $\mathcal{O}(m^{\frac{1}{2}})$. □

3 Further Work

In this section, we want to briefly discuss recent work that extends Kara et al.'s [8] algorithms and ideas.

Lu et al. [10] propose an exact algorithm for dynamic triangle counting whose performance depends on the *arboricity*[2] α of G. For any "nicely behaved" monotonic function Γ, it requires $O\left(\min\left\{\alpha m + m\log m, \left(\frac{m}{\Gamma(m)}\right)^2\right\}\right)$ space, $\tilde{O}(\min\{\alpha + \Gamma(m), \sqrt{m}\})$ amortized update time, and constant query time. By setting $\Gamma(m) = \sqrt{m}$, this reconstructs the result of Kara et al. [8]. For planar graphs, α is constant [7], therefore, their algorithm achieves $O(m \log m)$ space, $\tilde{O}(1)$ amortized update time, and $O(1)$ query time on this class of graphs.

Hanauer et al. [6] use the idea of ϵ-partitioning to count the number of occurences of all connected patterns consisting of four vertices. Different patterns have different time and space complexities. Paths of length 3, for example, require $O(\sqrt{m})$ time and linear space, whereas cliques of size 4 use linear time and quadratic space in m.

Dhulipala et al. [5] parallelize and extend the sequential algorithm of Kara et al. [8] to allow for batch updates, whereas the original algorithm only allowed for single edge updates at a time. Their algorithm requires $O(\Delta\sqrt{\Delta + m})$ amortized work and $\mathcal{O}(\log^*(\Delta + m))$[3] depth[4] with high probability, where Δ is the number of edge insertions and deletions, matching the original update time for $\Delta = 1$, but allowing for parallel execution of updates.

[2]The *arboricity* of a graph is the minimum number of spanning forests required s.t. their union is equal to the graph. It is bounded from above by $\mathcal{O}(\sqrt{m})$, but it can be significantly smaller. See, e.g., [7]

[3]$\log^*$ is the iterated logarithm. It is smaller than five for all inputs of size $\leq 2^{65536}$.

[4]Depth refers to the longest sequential dependence in the computation. Given an infinite number of processors, this is identical to the processing time.

4 Conclusion

In this paper, we have adapted the exact, dynamic, and sequential algorithm for maintaining the triangle count and enumeration of all triangles by Kara et al. [8] for general graphs using methods and concepts of graph theory. Their idea of partitioning nodes based on their degree is often seen in dynamic algorithms. We have also briefly discussed recent developments of algorithms that either extend their work or use the idea of degree-based node partitioning for similar problems.

References

[1] Josh Alman and Virginia Vassilevska Williams. "A refined laser method and faster matrix multiplication". In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 522–539.

[2] Noga Alon, Raphael Yuster, and Uri Zwick. "Finding and counting given length cycles". In: *Algorithmica* 17.3 (1997), pp. 209–223.

[3] Laurent Bulteau et al. "Triangle counting in dynamic graph streams". In: *Algorithmica* 76.1 (2016), pp. 259–278.

[4] Luciana S Buriol et al. "Counting triangles in data streams". In: *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2006, pp. 253–262.

[5] Laxman Dhulipala et al. "Parallel batch-dynamic k-clique counting". In: *Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM. 2021, pp. 129–143.

[6] Kathrin Hanauer, Monika Henzinger, and Qi Cheng Hua. "Fully dynamic four-vertex subgraph counting". In: *arXiv preprint arXiv:2106.15524* (2021).

[7] Frank Harary. *Graph theory*. Addison-Wesley, 1991. ISBN: 978-0-201-02787-7.

[8] Ahmet Kara et al. "Maintaining triangle queries under updates". In: *ACM Transactions on Database Systems (TODS)* 45.3 (2020), pp. 1–46.

[9] Konstantin Kutzkov and Rasmus Pagh. "Triangle counting in dynamic graph streams". In: *Scandinavian Workshop on Algorithm Theory*. Springer. 2014, pp. 306–318.

[10] Shangqi Lu and Yufei Tao. "Towards Optimal Dynamic Indexes for Approximate (and Exact) Triangle Counting". In: *24th International Conference on Database Theory (ICDT 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2021.

[11] Andrew McGregor, Sofya Vorotnikova, and Hoa T Vu. "Better algorithms for counting triangles in data streams". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2016, pp. 401–411.

[12] Hung Q Ngo et al. "Worst-case optimal join algorithms". In: *Journal of the ACM (JACM)* 65.3 (2018), pp. 1–40.

[13] Thomas Schank and Dorothea Wagner. "Approximating clustering coefficient and transitivity." In: *Journal of Graph Algorithms and Applications* 9.2 (2005), pp. 265–275.

[14] Charalampos E Tsourakakis, Mihail N Kolountzakis, and Gary L Miller. "Triangle Sparsifiers." In: *J. Graph Algorithms Appl.* 15.6 (2011), pp. 703–726.

[15] Todd L. Veldhuizen. "Triejoin: A Simple, Worst-Case Optimal Join Algorithm". In: *ICDT*. 2014.