

A Note on Improved Results for One Round Distributed Clique Listing

Quanquan C. Liu*

Abstract

In this note, we investigate listing cliques of arbitrary sizes in bandwidth-limited, dynamic networks. The problem of detecting and listing triangles and cliques was originally studied in great detail by Bonne and Censor-Hillel (ICALP 2019). We extend this study to dynamic graphs where more than one update may occur as well as resolve an open question posed by Bonne and Censor-Hillel (2019). Our algorithms and results are based on some simple observations about listing triangles under various settings and we show that we can list larger cliques using such facts. Specifically, we show that our techniques can be used to solve an open problem posed in the original paper: we show that detecting and listing cliques (of any size) can be done using $O(1)$ -bandwidth after one round of communication under node insertions and node/edge deletions. We conclude with an extension of our techniques to obtain a small bandwidth 1-round algorithm for listing cliques when more than one node insertion/deletion and/or edge deletion update occurs at any time.

1 Introduction

Detecting and listing subgraphs under limited bandwidth conditions is a fundamental problem in distributed computing. Although the static version of this problem has been studied by many researchers in the past [ACKL17, CCGL21, CGL20, CPSZ21, CPZ19, CS19, CS20, DKO14, EFF⁺19, FGKO18, FMO⁺17, GO17, HPZZ20, IG17, KR17, PRS18] in both the upper bound and lower bound settings, the dynamic version of such problems often require different techniques. Triangle and clique listing are problems where every occurrence of a triangle or clique in the graph is listed by at least one node in the triangle or clique. More specifically, the question we seek to answer is: given a change in the topology of the graph under one or more updates, can we accurately list all cliques in the updated graph? For certain settings, this question has an easy answer. For example, in the LOCAL model where communication is synchronous and error-free and messages can have unrestricted size, it is trivial for any node to list all k -cliques adjacent to it after any set of edge insertion/deletion and/or node insertion/deletion. In the LOCAL model, every node would broadcast the entirety of its adjacency list to all its neighbors each round. Thus, each node can reconstruct the edges between its neighbors from these messages and it is easy to solve the clique listing problem using the reconstructed neighborhood.

In the traditional CONGEST model, messages are passed between neighboring nodes in synchronous rounds where each message has size $O(\log n)$. Detecting and listing triangles and cliques in the CONGEST model turn out to be much harder problems. This note focuses on triangle and clique *listing* for which a number of previous works provided key results. The summary of these results can be found in Table 1. In the table, Δ is the maximum degree in the input graph. All of these results focus on the *static* setting where the topology of the graph does not change.

Additional works also provide lower bounds under very small bandwidth settings [ACKL17, IG17, FGKO18, PRS18]. A detailed description of all the aforementioned results can be found in the recent comprehensive survey of Censor-Hillel [CH21].

In this paper, we focus on the *dynamic* setting for subgraph listing problems for which we are able to obtain 1-round algorithms that require $O(1)$ or $O(\log n)$ bandwidth. In the dynamic setting, edge and node updates are applied to an initial (potentially empty) graph. The updates change the topology of the graph.

*Northwestern University, quanquan@northwestern.edu

Problem	Rounds
Triangle Listing	$\tilde{O}(n^{3/4})$ [IG17]
	$\tilde{O}(n^{1/2})$ [CPZ19]
	$\tilde{O}(n^{1/3})$ [CS19]
Triangle Listing Lower Bound	$\tilde{\Omega}(n^{1/3})$ [PRS18, IG17]
Deterministic Triangle Listing	$n^{2/3+o(1)}$ [CS20]
	$O(\Delta/\log n + \log \log \Delta)$ [HPZZ20]
4-Clique Listing	$n^{5/6+o(1)}$ [EFF+19]
4-Clique Listing Lower Bound	$\tilde{\Omega}(n^{1/2})$ [CK20]
5-Clique Listing	$n^{73/75+o(1)}$ [EFF+19]
	$n^{3/4+o(1)}$ [CGL20]
k -Clique Listing	$\tilde{O}(n^{k/(k+2)})$ ($k \geq 4, k \neq 5$) [CGL20]
	$\tilde{O}(n^{1-2/k})$ [CCGL21]
	$n^{1-2/(k+o(1))}$ [CLV22]
k -Clique Listing Lower Bound	$\tilde{\Omega}(n^{1-2/k})$ [FGKO18]
	$\tilde{\Omega}(n^{1/2}/k)$ ($k \leq n^{1/2}$) [CK20]
	$\tilde{\Omega}(n/k)$ ($k > n^{1/2}$) [CK20]

Table 1: Results for subgraph listing problems in the CONGEST model.

All nodes in the initial graph know the graph’s complete topology. After the application of each round of updates, the nodes collectively must report an accurate list of the correct set of triangles or cliques. We build off the elegant results of [BC19] who define and investigate in great depth the question of detecting and listing triangles and cliques in dynamic networks. The model they present in their paper can capture real-world behavior such as nodes joining or leaving the network or communication links which appear or disappear between pairs of nodes at different points in time. In this paper, we make several observations about triangle listing which may be used to re-prove results provided in their paper as well as resolve an open question stated in their work. Our paper is structured as follows. First, we formally define the model in Section 2. Then, we summarize our results in Section 3. Section 4 gives our main result for listing k -cliques under node insertions/deletions and edge deletions. Section 5 describes our result for listing wedges under node deletions and edge insertions/deletions. Section 6 describes our result on triangle listing under batched updates, and Section 7 gives our result on clique listing under batched updates.

2 Preliminaries

The model we use in our paper is the same as the model used in [BC19]. For completeness, we restate their model as well as the problem definitions in this section.

The network we consider in this dynamic setting can be modeled as a sequence of graphs: (G_0, \dots, G_r) . The initial graph G_0 represents the starting state of the network. All nodes in the initial graph G_0 know the graph’s complete topology. Each subsequent graph in the sequence is either identical to the preceding graph or differs from it by a single topology change: either an edge insertion, edge deletion, node insertion (along with its adjacent edges), or node deletion (along with its adjacent edges). Later, we also consider graphs where each node may be adjacent to $O(1)$ multiple topology changes. We denote the neighbors of node v in the i -th graph by $N_i(v)$; we omit the subscript i when the current graph is obvious from context.

We assume the network is synchronized. In each round, each node can send to each one of its neighbors a message containing B bits where B is defined as the *bandwidth* of the network. The messages sent to different neighbors can be different. In this paper, we focus on problems that can be solved with bandwidth $B = O(1)$ or $B = O(\log n)$.

We assume that each node has a unique ID and each node in the network knows the IDs of all its neighbors. In other words, we assume that each node has an adjacency list containing the IDs of all its neighbors and knows when a new node (previously not in its adjacency list) becomes connected to it. Furthermore, we assume that the size of any ID is $O(\log n)$ in bits. Since in both [BC19] and our work, the number of vertices in the graph can change dynamically, we define n to be the number of vertices in the graph in the current

round.

Each round proceeds in three synchronous parts as follows.

1. At the start of each round, the topological change occurs. Nodes are inserted/deleted from adjacency lists and new communication links are established or destroyed between pairs of nodes. A node can determine whether an adjacent update occurred on an adjacent edge by comparing its current list of neighbors with its list of neighbors from the previous round. However, this also means that a node u cannot distinguish between the insertion of an edge (u, v) and the insertion of node v .
2. Then, nodes exchange messages using (potentially new) communication links. Nodes previously able to communicate may not continue to be able to communicate once the links are destroyed due to new deletions.
3. Finally, nodes receive messages and list the triangles or cliques for this round.

In this paper, we only deal with 1-round algorithms or algorithms where the output of each node in the graph following 1 round of communication after the last topological change is correct. We define the *1-round bandwidth complexity* of an algorithm to be the minimum bandwidth B for which such a 1-round algorithm exists. Our paper only deals with *deterministic* 1-round algorithms.

We state a subgraph is *created* if it is a subgraph that is created by a new edge in the current round. We state a subgraph is *destroyed* if it is listed by at least one node in the previous round but no longer exists in the current round. When a node is deleted, it can no longer send any messages or list any subgraphs in the round where it is deleted (because it no longer exists).

We solve the triangle and k -clique *listing* problem in $O(1)$ or $O(\log n)$ bandwidth in this paper. We denote a clique with s vertices by K_s . Specifically, the problem $\text{List}(H)$ is defined to be the problem where given a generic unlabeled graph H , each labeled subgraph of the current graph $G_i \in (G_0, G_1, \dots, G_r)$ that is isomorphic to H must be listed by at least one node. A node *lists* a subgraph if it lists the IDs of all nodes in the subgraph. Furthermore, every listed subgraph must be isomorphic to H . Naturally, all of our listing algorithms also apply to the *detection* setting where at least one node in the network detects the appearance of one or more H (the problem denoted by $\text{Detect}(H)$ in [BC19]). As shown in Observation 1 of [BC19], $B_{\text{Detect}(H)} \leq B_{\text{List}(H)}$, and we only discuss $\text{List}(H)$ in the following sections whose results also transfer to $\text{Detect}(H)$.

2.1 Batched Updates Model

We define a new model where more than one update can occur in each round. Specifically, for every node $v \in V_i$ in the current graph $G_i = (V_i, E_i)$, at most $O(1)$ updates are incident to v . This means that any node in $G_{i-1} = (V_{i-1}, E_{i-1})$ is incident to $O(1)$ new edge insertions and deletions in G_i . A node $u \in V_i \setminus V_{i-1}$ is a newly inserted node and *all* edges incident to u are newly inserted edges. Hence, u is incident to $O(1)$ edges when it is inserted. We denote the subgraph H listing problem in this model as $\text{BatchedList}(H)$.

Notably, we do not need the assumption that each newly inserted node v can tell which of its neighbors are also newly inserted nodes. This is because all edges incident to newly inserted nodes are newly inserted edges. Thus, a newly inserted node v can send its entire adjacency list to all its neighbors. An important part of our model is that all nodes, including the newly inserted nodes, are adjacent to $O(1)$ updates so newly inserted nodes can send its adjacency list in $O(\log n)$ bandwidth to all its neighbors.

3 Our Contributions

All of the algorithms in our paper are deterministic 1-round algorithms that use $O(1)$ or $O(\log n)$ bandwidth. Specifically, we show the following main results in this paper.

We first answer Open Question 4 posed in [BC19]. The previous algorithm given in [BC19] under this setting required $O(\log n)$ bandwidth.

Theorem 3.1. *The deterministic 1-round bandwidth complexity of listing cliques, $\text{List}(K_s)$, under node insertions and node/edge deletions is $O(1)$.*

Algorithm 1: Listing Cliques

```
1 Input: Update  $U$  which can be a node insertion, node deletion, or edge deletion.
2 Output: Each  $K_s$  is listed by at least one of its nodes and no set of  $s$  nodes which is not a  $K_s$  is
   wrongly listed.
3 Function ListCliques( $U$ )
4   if  $U$  adds  $u$  to  $N(v)$  then
5      $v$  sends NEW to all  $w \in N(v)$ .
6   if  $w$  receives NEW from  $v$  and  $u$  is added to  $N(w)$  then
7      $w$  starts listing new triangle  $\{u, v, w\}$ .
8   if  $U$  deletes  $u$  from  $N(v)$  then
9      $v$  sends DELETE to all  $w \in N(v)$ .
10    for  $w \in N(v)$  do
11      if  $v$  lists triangle  $\{u, v, w\}$  then
12         $v$  stops listing triangle  $\{u, v, w\}$ .
13  if  $v$  receives DELETE from  $u$  and  $w$  then
14    if  $v$  lists triangle  $\{u, v, w\}$  then
15       $v$  stops listing triangle  $\{u, v, w\}$ .
16  for all  $v \in V$  do
17    for every  $S \subseteq N(v)$  of  $s - 1$  neighbors do
18      if  $v$  lists  $\{v\} \cup A$  as a triangle for every subset of 2 nodes  $A = \{a, b\} \subseteq S$  then
19         $v$  lists  $S \cup \{v\}$  as a  $K_s$ .
20   $v$  stops listing every clique containing a destroyed triangle.
```

To prove this result, we also show a number of simple but new observations about triangle listing in [Section 4](#) which may prove to be useful for more future research in this area.

In addition to cliques, we give an algorithm for wedges, which has not been considered in previous literature. An *induced wedge* among a set of three nodes $\{u, v, w\}$ in the input graph is a path of length 2 that uses all three nodes, and the induced subgraph of the three nodes is not a cycle. We denote a wedge by Γ .

Theorem 3.2. *The deterministic 1-round bandwidth complexity of listing wedges, $\text{List}(\Gamma)$, under edge insertions and node/edge deletions, is $O(\log n)$.*

Censor-Hillel *et al.* [CHKS21] studied the setting of *highly* dynamic networks in which the number of topology changes per round is *unlimited*. In this setting, they showed $O(1)$ -amortized round complexity algorithms for k -clique listing and four and five cycle listing. In our 1-round CONGEST setting, we show the following two theorems when multiple updates occur in the same round. Our results are the first to consider more than one update in the 1-round, low bandwidth setting.

Theorem 3.3. *The deterministic 1-round bandwidth complexity of listing triangles, $\text{BatchedList}(K_3)$, when each node is incident to $O(1)$ updates, is $O(\log n)$ under node/edge insertions and node/edge deletions.*

Theorem 3.4. *The deterministic 1-round bandwidth complexity of listing cliques of size s , $\text{BatchedList}(K_s)$, when each node is incident to $O(1)$ updates, is $O(\log n)$ under node insertions and node/edge deletions.*

4 Clique Detection and Listing

In this section, we show our main result by providing a simple upper bound for the 1-round bandwidth complexity of $\text{List}(K_s)$ under node insertions and node/edge deletions. Specifically, we show that $\text{List}(K_s)$ has 1-round $O(1)$ -bandwidth complexity. This directly resolves Open Question 4 in [BC19]. We describe our algorithm below and give its pseudocode in [Algorithm 1](#). To begin as a simple warm-up illustration of our technique, we re-prove Theorem 3.3.3 of [BC19] using only the node insertion portion of the algorithm.

In [Algorithm 1](#), for every update which adds a node u to an adjacency list $N(v)$ ([Line 4](#)), v sends an $O(1)$ sized message to every neighbor $w \in N(v)$ ([Line 5](#)). If any neighbor w also gets u added to its adjacency list $N(w)$ ([Line 6](#)), then w lists triangle $\{u, v, w\}$. If instead u is deleted from $N(v)$ ([Line 8](#)), then v first sends DELETE to all its neighbors $w \in N(v)$ ([Line 9](#)). Furthermore, for every neighbor $w \in N(v)$, if v lists triangle $\{u, v, w\}$ ([Line 11](#)), then v stops listing $\{u, v, w\}$ ([Line 12](#)). If a node v receives a DELETE message from two of its neighbors ([Line 13](#)) and if it lists $\{u, v, w\}$ as a triangle ([Line 14](#)), then it stops listing $\{u, v, w\}$ ([Line 15](#)). Finally, after receiving all the messages for this round, every vertex $v \in V$ determines the subsets $S \subseteq N(v)$ of $s - 1$ of its neighbors ([Line 17](#)) where every subset of 2 nodes of $\{a, b\} \subseteq S$ including itself, $\{a, b\} \cup \{v\}$, is a triangle ([Line 18](#)); each such $S \cup \{v\}$ is a K_s and v lists it as a K_s . Note that [Algorithm 1](#) treats a node that is deleted and reinserted like a new node when it is reinserted (i.e. nodes do not distinguish between a completely new neighbor from one that is deleted and reinserted at a later time).

We now prove a set of useful observations which we use to prove our main theorem regarding $\text{List}(K_s)$. The first observation below states that any node inserted in round r will list every triangle containing it that is formed after its insertion. This is the crux of our analysis since this combined with our other observations states that every clique is listed by its earliest inserted node.

Observation 4.1. *Under node insertions and node/edge deletions, suppose v is inserted in round r and u is inserted in round $r' > r$. If $\{u, v, w\}$ is a triangle in round r' , then in round r' , u lists $\{u, v, w\}$ using $O(1)$ bandwidth.*

Proof. We prove this via induction. We assume our observation holds for the j -th node insertion and prove it for the $(j + 1)$ -st insertion. The observation trivially holds for the initial graph since all nodes know the topology of the initial graph and thus lists all triangles they are part of. Let v be a node inserted in round $r \leq j$ and u be a node inserted in round $j + 1$. If u forms a triangle with v and another node w , when node u is inserted, it must contain an edge to both v and w and edge $\{v, w\}$ must already exist in the graph. Edge $\{v, w\}$ must already exist in the graph since edges are only inserted as a result of node insertions; hence, if an edge between v and w exists it must have been inserted when either v or w was inserted (or it existed in the initial graph), whichever one was inserted later. When a node receives a new neighbor, it sends NEW to all its neighbors indicating it received a new neighbor ([Line 5](#) of [Algorithm 1](#)). Since both v and w are adjacent and send NEW to each other, v receives NEW from w and knows that u must be its new neighbor. Thus, v lists the triangle containing w and u . Because this holds for all vertices inserted in round $r \leq j$, we proved that every node inserted in round $r \leq j$ correctly lists every triangle created in round $j + 1$ that contains it.

We now show that v does not incorrectly list a set of three nodes $\{u, v, w\}$ as a triangle when the set is not a triangle. This means that we show on an edge or node deletion, v stops listing triangles that are destroyed. We first show that any node which lists a triangle will also successfully list its deletion after an edge deletion. First, if an edge deletion removes a neighbor of v that is part of a triangle that v lists, then v stops listing this triangle ([Line 12](#)). Node v also sends DELETE to all its neighbors ([Line 9](#)). Suppose the edge deletion happens between u and v and w lists triangle $\{u, v, w\}$. Then, w receives DELETE from u and v in round $j + 1$ and knows that edge $\{u, v\}$ is deleted; w then stops listing $\{u, v, w\}$ ([Lines 13 to 15](#)). Hence, after an edge deletion in round $j + 1$, all nodes which lists a triangle destroyed by the deletion stops listing the triangle. What remains to be shown is that a node which lists a triangle can successfully list its deletion after a node deletion. When a triangle is destroyed due to a node deletion, one of its nodes must have been deleted. Hence, any node u which lists a triangle $\{u, v, w\}$ and sees the deletion of one of its two neighbors from its adjacency list will stop listing it. \square

Observation 4.2 ([\[BC19\]](#)). *If a node lists all the triangles containing it within a clique of size $k \geq 3$, then the node lists the clique.*

Proof. This observation was made in [\[BC19\]](#). If a node lists all triangles containing it within a clique, then it lists all edges between the nodes in each of these triangles. From this information, it can compute and list every clique that may be formed from the set of nodes in these triangles. \square

Theorem 4.3 (Theorem 3.3.3 [\[BC19\]](#)). *The deterministic 1-round bandwidth complexity of $\text{List}(K_s)$ under node insertions is $O(1)$.*

Proof. By [Observation 4.1](#), under node insertions, any node v inserted in round r lists every triangle containing it and a node u inserted in round $r' > r$ in 1-round using $O(1)$ -bandwidth, deterministically. Then, suppose K_s is formed in round j after the j -th node insertion. Each node of K_s in the initial graph lists all subsequent triangles formed in K_s from node insertions (or the first inserted node can do this if no nodes in K_s were present in the initial graph). By [Observation 4.2](#), this node lists K_s in round j using $O(1)$ bandwidth. \square

Corollary 4.4. *Given node insertions, the i -th node v inserted for clique K_s lists the smaller K_{s-i+1} clique of K_s composed of nodes inserted after v .*

Proof. By [Observation 4.1](#), each node v lists all triangles containing v and incident to nodes inserted after it. Hence, each node lists all incident cliques with nodes inserted after it. \square

Observation 4.5. *If a node v lists a clique of size $k \geq 3$, then v lists all triangles containing v in the clique.*

Proof. This observation can be easily shown: if a node lists a clique, then it lists all nodes contained in the clique by definition. Then, because the node knows that such a clique exists, it can find all combinations of 3 nodes in the clique and list such combinations as all triangles in the clique. \square

Observation 4.6. *Any node/edge deletion in a clique results in (at least) one adjacent triangle deletion (where the triangle is composed of nodes in the clique) for every node in the clique.*

Proof. A clique consists of a set of triangles which contains all subsets of 3 vertices from the clique. Suppose the edge deletion occurs between vertices u and v , then any other vertex $w \neq u, v \in C$, where C is the clique, must be adjacent to u and v , and hence formed a triangle with u and v . Since the edge between u and v is destroyed, w cannot form a triangle with u and v , and hence, a triangle adjacent to w is destroyed. \square

Given the above observations and the relevant lemmas and theorems from [\[BC19\]](#), we are now ready to prove the main theorem of this section which resolves Open Question 4 of [\[BC19\]](#).

Theorem 3.1. *The deterministic 1-round bandwidth complexity of listing cliques, $\text{List}(K_s)$, under node insertions and node/edge deletions is $O(1)$.*

Proof. To begin, we stipulate that no node lists a clique if it did not list the clique previously after receiving a deletion message. [Observation 4.1](#) shows that every node v which is inserted in round r lists every triangle containing it and a node u inserted in round $r' > r$. Using this observation, we handle the updates in the following way (reiterating the strategy employed by our key observations). First, we show that any created clique is listed by at least one of the nodes it contains. By [Observation 4.1](#) and [Observation 4.2](#), the node that is inserted first in a K_s clique lists the K_s clique that is created by node insertions.

We now show the crux of our proof that each node which lists K_s also successfully stops listing it after it is destroyed. As proven previously in [Observation 4.1](#), any node which lists a triangle stops listing it when it is destroyed. Given any edge deletion or node deletion which destroys a clique C , it must destroy at least one triangle adjacent to every node in the clique C by [Observation 4.6](#). Every node which lists the clique lists all triangles containing it in the clique by [Observation 4.5](#). Then, every node v which lists a clique will know if one of its listed triangles is destroyed; then v will also know of the clique that is destroyed if a triangle that makes up the clique is destroyed.

Finally, to conclude our proof, we explain one additional scenario that may occur: a clique which is destroyed may be added back again later on. Any clique which is destroyed due to an edge deletion cannot be added again unless a node deletion followed by another node insertion occurs since we are only allowed edge insertions associated with node insertions. Thus, by [Corollary 4.4](#), the earliest node(s) (if such nodes existed in the initial graph) in this newly formed clique lists this clique.

Hence, we have proven that any created clique is listed by at least one of its nodes and that any node which lists the clique also stops listing it after it is destroyed, concluding our proof of our theorem. The bandwidth is $O(1)$ since each node sends either no message, NEW, or DELETE to each of its neighbors each round. \square

Algorithm 2: Listing Wedges

```
1 Input: Update  $U$  which can be a node deletion, edge insertion, or edge deletion.
2 Output: Each induced wedge is listed by at least one of its nodes and no set of three nodes which
   is not a wedge is wrongly listed as a wedge.
3 Function ListWedges( $U$ )
4   if  $U$  adds  $u$  to  $N(v)$  then
5      $v$  sends (NEW,  $ID_u$ ) to all  $w \in N(v)$ .
6     for every wedge  $(v, x, u)$  that  $v$  lists do
7        $v$  stops listing wedge  $(v, x, u)$ .
8        $v$  starts listing triangle  $\{v, x, u\}$ .
9   else if  $U$  deletes  $u$  from  $N(v)$  then
10     $v$  sends (DELETE,  $ID_u$ ) to all  $w \in N(v)$ .
11    for every triangle  $\{v, x, u\}$  that  $v$  lists do
12       $v$  starts listing new wedge  $(v, x, u)$ .
13       $v$  stops listing triangle  $\{v, x, u\}$ .
14    for every wedge  $(x, v, u)$  and/or  $(u, v, x)$  that  $v$  lists do
15       $v$  stops listing  $(u, v, x)$  and/or  $(x, v, u)$ .
16  if node  $x$  receives (NEW,  $ID_u$ ) from  $v$  then
17    if  $x$  is not adjacent to  $u$  then
18       $x$  starts listing new wedge  $(x, v, u)$ .
19    if  $x$  lists wedge  $(v, x, u)$  then
20       $x$  stops listing wedge  $(v, x, u)$ .
21       $x$  starts listing triangle  $\{v, x, u\}$ .
22  if  $x$  receives (DELETE,  $ID_u$ ) from  $v$  then
23    if  $x$  lists wedge  $(x, v, u)$  then
24       $x$  stops listing wedge  $(x, v, u)$ .
25    if  $x$  lists triangle  $\{x, v, u\}$  then
26       $x$  starts listing new wedge  $(v, x, u)$ .
27       $x$  stops listing triangle  $\{v, x, u\}$ .
```

5 Wedge Listing

An *induced wedge* (u, v, w) in the input graph $G = (V, E)$ is a path of length 2 where edges (u, v) and (v, w) exist and no edge exists between u and w in the induced subgraph consisting of nodes $\{u, v, w\} \subseteq V$. We denote a wedge by Γ . In this section, we provide an algorithm listing *induced* wedges. Listing non-induced wedges is a much simpler problem since each node can simply list pairs of its adjacent neighbors without knowing whether an edge exists between each pair. For simplicity, from here onward, we say “wedge” to mean induced wedge. [Theorem 3.2](#) is the main theorem in this section for listing wedges. We give the pseudocode for the algorithm used in the proof of this theorem in [Algorithm 2](#). Our algorithm uses listing triangles as a subroutine since an induced wedge can be formed from an edge deletion to a triangle. Listing wedges may be useful as a subroutine in future work that proves additional bounds for listing other small subgraphs using small bandwidth.

We describe our algorithm below. All messages are sent *in the same round*; multiple messages sent between u and v are concatenated with each other and sent as one message. On an update U , in [Algorithm 2](#), each node sends $O(1)$ bits, indicating whether a neighbor is deleted or added to their neighbors. Let us denote these bits by DELETE and NEW, respectively. Furthermore, they send the ID of the neighbor that is deleted or inserted. Let this ID be ID_u for node u . This procedure is shown in [Lines 5 and 10](#) in [Algorithm 2](#) and all of the message sending is done in 1 round. An edge insertion can form a triangle; thus, in the same round as the (NEW, ID_u) message, node v also starts listing the new triangle $\{v, x, u\}$ ([Lines 7 and 8](#)) and stops listing the wedge. The node/edge deletion can destroy a wedge ([Line 14](#)); in which case, the node stops listing the wedge ([Line 15](#)). Similarly, for an edge/node deletion that destroys a triangle, v starts listing wedge (v, x, u) for every triangle $\{v, x, u\}$ that v lists ([Lines 12 and 13](#)).

Any node x that receives (NEW, ID_u) from a neighbor v (Line 16) lists (x, v, u) as a new wedge (Line 18) if it is not adjacent to u (Line 17). If x lists wedge (v, x, u) (Line 19), then x stops listing the wedge (Line 20) and starts listing triangle $\{v, x, u\}$ (Line 21).

Any node x that receives (DELETE, ID_u) from a neighbor v (Line 22) first checks if a wedge it lists is destroyed. If x lists wedge (x, v, u) , then it stops listing wedge (x, v, u) since it is destroyed (Line 24). Then, it checks if the deletion destroys a triangle it lists. If it lists triangle $\{x, v, u\}$, then it lists a new wedge (v, x, u) since a destroyed triangle creates a new wedge (Lines 26 and 27) and stops listing the triangle.

Theorem 3.2. *The deterministic 1-round bandwidth complexity of listing wedges, $\text{List}(\Gamma)$, under edge insertions and node/edge deletions, is $O(\log n)$.*

Proof. We prove via induction that this algorithm achieves the following guarantees

- (a) any wedge is listed by at least one node
- (b) and no sets of three nodes that is not a wedge is incorrectly listed as a wedge

after any edge insertion or node/edge deletion. In the base case, the graph is either the input graph and every wedge in the graph is listed by at least one node or the graph is empty. We assume that at step j , each wedge in the graph is listed by some node in the wedge. We show that all wedges formed in round $j+1$ by some update is listed by at least one node in the wedge; then, we show that each wedge that is destroyed in round $j+1$ is no longer listed by any node. We prove this by casework over the type of update:

- Edge insertion: Given an edge insertion $\{u, v\}$, node v sees some node u added to its adjacency list and sends (NEW, ID_u) to all its neighbors in $N(v)$ (similarly for u). Suppose $x \in N(v)$ is a neighbor of v but not of u . Then, x can distinguish between triangle $\{x, v, u\}$ and wedge (x, v, u) by seeing whether u is in its adjacency list (i.e. whether edge $\{x, u\}$ exists). Hence, x lists wedge (x, v, u) .

An edge insertion can cause a wedge to be destroyed if it forms a triangle from a wedge. Suppose (x, v, u) is a wedge that becomes a triangle due to edge insertion $\{x, u\}$. If x lists the wedge, then x knows $\{x, v, u\}$ is now a triangle because it sees u added to its adjacency list. The same argument holds for u . If v lists the wedge, then v receives (NEW, ID_x) and (NEW, ID_u) and knows that $\{x, v, u\}$ is now a triangle and stops listing it as a wedge.

- Edge deletion: Suppose an edge deletion $\{u, v\}$ destroys wedge (x, v, u) . Since u and v can see that the other is no longer a neighbor, if they could list wedge (x, v, u) , they now know that (x, v, u) has been destroyed and stops listing it. Node v sends to x the tuple (DELETE, ID_u) using $O(\log n)$ bandwidth and 1 round of communication. Hence, after this one round of communication, x , if it can list (x, v, u) , now knows (x, v, u) has been destroyed and stops listing it.

We now show that if an edge deletion destroys a triangle, then at least one of its nodes lists the new wedge. Under our current set of update operations, any triangle is formed after 3 edge insertions. After the first two edge insertions, at least one node x lists the wedge that is formed by our argument provided above for edge insertions. Then, after the third insertion, x lists the triangle formed (and stops listing the wedge). Suppose without loss of generality that node x lists triangle $\{x, v, u\}$. If $\{x, u\}$ is deleted, x sees u removed from its adjacency list and lists (x, v, u) as a new wedge. The same argument holds for edge deletion (x, v) . If instead (u, v) is deleted, then node x receives (DELETE, ID_v) and (DELETE, ID_u) and lists (v, x, u) as a new wedge.

- Node deletion: Suppose wlog u is deleted and there exists some wedge containing u, v , and w . Then, if u is deleted and the wedge consists of edges $\{v, u\}$ and $\{u, w\}$, then both v and w notice that u has been removed from their adjacency lists. Hence, if either of the nodes listed the wedge, they would know it is destroyed and stops listing it. If the wedge consists of edges $\{u, v\}$ (resp. $\{u, w\}$) and $\{v, w\}$, then w would know of the deletion of edge $\{u, v\}$ (resp. v of edge $\{u, w\}$) after receiving messages since v (resp. w) sent to all its neighbors (DELETE, ID_u) .

Hence, in step $j+1$ after the $j+1$ -st synchronous round, all wedges formed in step $j+1$ is listed by at least one vertex. The bandwidth is $O(\log n)$ since the IDs of vertices are $O(\log n)$ sized. \square

Algorithm 3: Batched Listing Triangles

```
1 Input: Updates  $\mathcal{U}$  which is a set of updates consisting of node insertions, node deletions, edge
   insertions, and/or edge deletions.
2 Output: Each triangle is listed by at least one of its nodes and no set of 3 nodes which is not a
   triangle is wrongly listed.
3 Function BatchedListTriangles( $\mathcal{U}$ )
4   if  $\mathcal{U}$  adds node(s)  $u \in I_v$  to  $N(v)$  then
5      $v$  sends (NEW,  $\{ID_u \mid u \in I_v\}$ ) to all  $w \in N(v)$ .
6   if  $w$  receives (NEW,  $S_v$ ) from  $v$  then
7     for  $ID_u \in S_v$  do
8       if  $u \in N(w)$  then
9          $w$  starts listing new triangle  $\{u, v, w\}$ .
10  if  $\mathcal{U}$  deletes node(s)  $u \in D_v$  from  $N(v)$  then
11     $v$  sends (DELETE,  $\{ID_u \mid u \in D_v\}$ ) to all  $w \in N(v)$ .
12  for  $u \in D_v$  do
13    for  $w \in N(v) \cup D_v$  do
14      if  $v$  lists triangle  $\{u, v, w\}$  then
15         $v$  stops listing triangle  $\{u, v, w\}$ .
16  if  $w$  receives (DELETE,  $S_v$ ) from  $v$  then
17    for  $ID_u \in S_v$  do
18      for all triangles  $\{u, v, w\}$  that  $w$  lists do
19         $w$  stops listing triangle  $\{u, v, w\}$ .
```

6 Batched Triangle Listing

In this section, we extend our algorithms in the previous sections to handle batches of more than one update. Our results are in the batched model (formally defined in [Section 2](#)) where in each batch of updates, $O(1)$ updates occur *adjacent to any node in the graph*. Such a model is realistic since often many updates can occur in total over an entire real-world network (such a social network) but the updates adjacent to each node in the network are often few in number.

We show that using our techniques above, we can also perform $\text{List}(K_3)$ in 1-round using $O(\log n)$ bandwidth. When provided a batch of updates, each node with one or more new neighbors sends the ID of the new neighbor(s) to all its neighbors. Furthermore, each node sends the IDs of all nodes deleted from its adjacency list to all its neighbors. We show that this simple algorithm allows us to perform $\text{List}(K_3)$ in one round under any type of update. The pseudocode for this algorithm is given in [Algorithm 3](#). All messages are sent in the same round in [Algorithm 3](#) and multiple messages sent (in the pseudocode) between two nodes are concatenated into the same message.

Theorem 3.3. *The deterministic 1-round bandwidth complexity of listing triangles, $\text{BatchedList}(K_3)$, when each node is incident to $O(1)$ updates, is $O(\log n)$ under node/edge insertions and node/edge deletions.*

Proof. For any inserted edge $\{u, v\}$, node u sends ID_v to all its neighbors (and similarly for node v). Any neighbor w of both u and v (and where neither u or v are deleted from its adjacency list) lists the new triangle $\{u, v, w\}$. Edges $\{u, w\}$ and/or $\{v, w\}$ may also be new edges and the proof still holds.

Now, we consider node insertions. An inserted node, u , creates a triangle if it is adjacent to two newly inserted edges, $\{u, v\}$, $\{u, w\}$, and the edge $\{v, w\}$ exists in the graph or is newly inserted. If $\{v, w\}$ already exists, then v sends ID_u to w and w sends ID_u to v ; thus, both v and w now list triangle $\{u, v, w\}$. If $\{v, w\}$ is newly inserted, then v and w would still send ID_u to each other and also send each other's IDs to u . Thus, all three nodes now list the triangle.

Finally, we consider node/edge deletions. For any destroyed triangle $\{u, v, w\}$, suppose wlog that u lists it. Either u is adjacent to an edge deletion, in which case, u stops listing $\{u, v, w\}$; or, u is still adjacent to both v and w and edge $\{v, w\}$ is deleted. In the second case, v and w both send u each other's ID and node u stops listing $\{u, v, w\}$. A node deletion is incident to all remaining nodes of the triangle and so the

Algorithm 4: Batched Listing Cliques

```
1 Input: Updates  $\mathcal{U}$  which is a set of updates consisting of node insertions, node deletions, and/or
   edge deletions.
2 Output: Each  $K_s$  is listed by at least one of its nodes and no set of  $s$  nodes which is not a  $K_s$  is
   wrongly listed.
3 Function BatchedListCliques( $\mathcal{U}$ )
4   Run BatchedListTriangles( $\mathcal{U}$ ) (Algorithm 3).
5   for all  $v \in V$  do
6     for every  $S \subseteq N(v)$  of  $s - 1$  neighbors do
7       if  $v$  lists  $\{v\} \cup A$  as a triangle for every subset of 2 nodes  $A = \{a, b\} \subseteq S$  then
8          $v$  lists  $S \cup \{v\}$  as a  $K_s$ .
9    $v$  stops listing every clique containing a destroyed triangle.
```

remaining nodes stop listing the triangle.

Since each node is adjacent to $O(1)$ updates and sends a message of size equal to the number of adjacent updates times the size of each node's ID (where the size of each ID is $O(\log n)$), the total bandwidth used is $O(\log n)$. \square

7 Batched Clique Listing

A slightly modified algorithm allows us to perform $\text{List}(K_s)$ under the same constraints; however, the proof is more involved than the K_3 case but uses the same concepts we developed in the previous sections. The modification computes the cliques using the triangles listed after a batch of updates similar to Algorithm 1.

Unlike the case for triangles, here, as in the case for counting cliques under single updates, Theorem 3.1, we cannot handle edge insertions in $O(\log n)$ bandwidth. We give the new pseudocode for this procedure in Algorithm 4. This part of the procedure is identical to that given in Algorithm 1.

To prove the main theorem in this section, Theorem 3.4, we only need to prove a slightly different version of Observation 4.2 that still holds in this setting. Namely, we modify Observation 4.2 so that it holds for triangles created by two or more updates in the same round.

Observation 7.1. *Under node insertions and node/edge deletions, suppose v is inserted in round r and u is inserted in round $r' \geq r$. If $\{u, v, w\}$ is a triangle in round r' , then in round r' , u lists $\{u, v, w\}$ using $O(1)$ bandwidth.*

Proof. The key difference between this observation and Observation 4.2 is that the observation holds when u is inserted in the same round r as v . Suppose triangle $\{u, v, w\}$ is created in round r' . We first consider the case when $r' > r$. In this case, w sends ID_u to v and v sees u in its adjacency list. Thus, v lists triangle $\{u, v, w\}$. Now, suppose $r' = r$. In this case, w still sends ID_u to v and v sees u in its adjacency list; thus, v lists $\{u, v, w\}$.

There are multiple scenarios for edge deletions that destroy triangle $\{u, v, w\}$. First, if one edge, $\{u, v\}$, is deleted, then u and v send each other's IDs to w in deletion messages. Thus, w stops listing $\{u, v, w\}$ if it previously listed $\{u, v, w\}$. If two edges are deleted, then, every node is adjacent to at least one deletion and will stop listing the triangle. A node deletion causes at least one adjacent edge to be deleted from every remaining node in the triangle; thus, any remaining node(s) will stop listing the triangle. \square

Observation 7.2. *Under node insertions and node/edge deletions, if triangle $\{u, v, w\}$ is created in round r from two or more node insertions, then all three nodes in the triangle lists it.*

Proof. Without loss of generality, suppose u and v are inserted in round r . Then, u and v sees each other in their adjacency lists. Node w sends ID_u and ID_v to both u and v . Hence, both u and v lists triangle $\{u, v, w\}$. Node w receives ID_u from v and ID_v from u and sees both u and v in its adjacency list. Hence, node w lists $\{u, v, w\}$. When all three nodes are inserted in the same round, each node receive the other nodes' IDs via messages and successfully lists the triangle. The remaining parts of this proof is identical to the second part of the proof of Observation 7.1. \square

Theorem 3.4. *The deterministic 1-round bandwidth complexity of listing cliques of size s , $\text{BatchedList}(K_s)$, when each node is incident to $O(1)$ updates, is $O(\log n)$ under node insertions and node/edge deletions.*

Proof. The key to this proof, as in the case for one update, is that there exists one node that lists all triangles incident to it in a new clique and hence lists the clique. By [Observation 7.1](#), the node inserted in the earliest round for each K_s lists it. If all nodes in a K_s are inserted in the current round, then by [Observation 7.2](#) and [Observation 4.2](#), all nodes in the clique lists it.

We now show that each node which lists a clique will stop listing it when it is destroyed. By [Observation 4.6](#), at least one triangle is destroyed for every node in a clique C if an edge deletion destroys C . Then, a triangle is destroyed by either an edge deletion or a node deletion (or both). For a node deletion v , an edge incident to every other node in any triangle containing v will be deleted. Thus, each node listing the triangle containing v will stop listing it when v is deleted. For any edge deletion $\{u, v\}$ that destroys a triangle $\{u, v, w\}$, if edges $\{u, w\}$ and $\{v, w\}$ are not deleted, then w receives a delete message from u and v listing each other's ID. Then, w knows that triangle $\{u, v, w\}$ is destroyed and stops listing it. We know every node v that lists C lists all of the triangles composed of nodes in C that are incident to v . We just showed that every node that lists a triangle successfully stops listing it when it is destroyed. Then, it follows that each node which lists C stops listing it when it is destroyed.

Since we are guaranteed $O(1)$ incident updates to every node, by [Algorithm 4](#), each node sends $O(1)$ IDs for each batch of updates and the size of each message is still $O(\log n)$. \square

8 Open Questions

We hope the observations we make in this paper will be useful for future work. The open question we find the most interesting regarding work in this area are techniques to list other induced subgraphs that are not cliques in one round under very small bandwidth. A preliminary look at listing k -paths and k -cycles in 1-round under $O(1)$ -bandwidth proves to be impossible if node deletions are allowed due to the following reason. Suppose we are given a subgraph H with radius greater than 2 and suppose v is currently listing H . Then a node deletion of a node u that destroys the subgraph, where u is at a distance greater than 2 from v , will not be detected by v in one round. However, perhaps results could be found for other types of subgraphs and if there are no node/edge deletions. Another interesting open question is whether these results could be extended to *arbitrarily* large number of edge/node updates, specifically extending our results given in the batched setting.

Acknowledgements

We are very grateful to our anonymous reviewers whose suggestions greatly improved the presentation of our results. In particular, we thank one anonymous reviewer for the simple argument that k -path and k -cycle listing in one round is difficult for radius greater than 2.

References

- [ABFL18] James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão, editors. *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, volume 95 of *LIPICs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
- [ACKL17] Amir Abboud, Keren Censor-Hillel, Seri Khoury, and Christoph Lenzen. Fooling views: A new lower bound technique for distributed computations under congestion. *CoRR*, abs/1711.01623, 2017.
- [BC19] Matthias Bonne and Keren Censor-Hillel. Distributed detection of cliques in dynamic networks. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, pages 132:1–132:15, 2019.

- [CCGL21] Keren Censor-Hillel, Yi-Jun Chang, François Le Gall, and Dean Leitersdorf. Tight distributed listing of cliques. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2878–2891. SIAM, 2021.
- [CGL20] Keren Censor-Hillel, François Le Gall, and Dean Leitersdorf. On distributed listing of cliques. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 474–482. ACM, 2020.
- [CH21] Keren Censor-Hillel. Distributed Subgraph Finding: Progress and Challenges. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:14, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [CHKS21] Keren Censor-Hillel, Victor I. Kolobov, and Gregory Schwartzman. Finding subgraphs in highly dynamic networks. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '21*, page 140–150, New York, NY, USA, 2021. Association for Computing Machinery.
- [CK20] Artur Czumaj and Christian Konrad. Detecting cliques in CONGEST networks. *Distributed Comput.*, 33(6):533–543, 2020.
- [CLV22] Keren Censor-Hillel, Dean Leitersdorf, and David Vulakh. Deterministic near-optimal distributed listing of cliques. In *PODC*, pages 271–280. ACM, 2022.
- [CPSZ21] Yi-Jun Chang, Seth Pettie, Thatchaphol Saranurak, and Hengjie Zhang. Near-optimal distributed triangle enumeration via expander decompositions. *J. ACM*, 68(3):21:1–21:36, 2021.
- [CPZ19] Yi-Jun Chang, Seth Pettie, and Hengjie Zhang. Distributed triangle detection via expander decomposition. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 821–840. SIAM, 2019.
- [CS19] Yi-Jun Chang and Thatchaphol Saranurak. Improved distributed expander decomposition and nearly optimal triangle enumeration. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 66–73. ACM, 2019.
- [CS20] Yi-Jun Chang and Thatchaphol Saranurak. Deterministic distributed expander decomposition and routing with applications in distributed derandomization. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 377–388. IEEE, 2020.
- [DKO14] Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 367–376. ACM, 2014.
- [EFF⁺19] Talya Eden, Nimrod Fiat, Orr Fischer, Fabian Kuhn, and Rotem Oshman. Sublinear-Time Distributed Algorithms for Detecting Small Cliques and Even Cycles. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:16, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [FGKO18] Orr Fischer, Tzlil Gonen, Fabian Kuhn, and Rotem Oshman. Possibilities and impossibilities for distributed subgraph detection. In Christian Scheideler and Jeremy T. Fineman, editors, *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 153–162. ACM, 2018.

- [FMO⁺17] Pierre Fraigniaud, Pedro Montealegre, Dennis Olivetti, Ivan Rapaport, and Ioan Todinca. Distributed subgraph detection. *CoRR*, abs/1706.03996, 2017.
- [GO17] Tzlil Gonen and Rotem Oshman. Lower bounds for subgraph detection in the CONGEST model. In Aspnes et al. [ABFL18], pages 6:1–6:16.
- [HPZZ20] Dawei Huang, Seth Pettie, Yixiang Zhang, and Zhijun Zhang. The communication complexity of set intersection and multiple equality testing. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, page 1715–1732, USA, 2020. Society for Industrial and Applied Mathematics.
- [IG17] Taisuke Izumi and François Le Gall. Triangle finding and listing in CONGEST networks. In Elad Michael Schiller and Alexander A. Schwarzhmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 381–389. ACM, 2017.
- [KR17] Janne H. Korhonen and Joel Rybicki. Deterministic subgraph detection in broadcast CONGEST. In Aspnes et al. [ABFL18], pages 4:1–4:16.
- [PRS18] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the distributed complexity of large-scale graph computations. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 405–414, New York, NY, USA, 2018. Association for Computing Machinery.