I. STREAMING GRAPH ALGORITHM INSTANCES

Based on different state updating strategies, GraphFlow provides efficient and effective graph updates. GraphFlow is a general model to support different graph algorithms. Because of the space limitation, here, we only select the Triangle Count algorithm as an example to illustrate the application of our GraphFlow model.

A. GraphFlow-based Triangle Count (GTC)

TC algorithm is used to count the number of different triangles in undirected graphs. Following the GraphFlow model, here is a design of State, Event, and Transform for GraphFlow-based Triangle Count (GTC) algorithm:

State: The state of a graph consists of each vertex's adjacent vertices information, i.e., $State = \{f_1, f_2, \cdots, f_n\}$, where n is the number of vertices in the graph. We define the factor of a node v_i for TC as (3):

$$f_i = (v_i, A_i, t_i),\ 1 \leq \mathrm{i} \leq \mathrm{n} \tag{3}$$

where A_i is the set of adjacent vertices of v_i, and t_i is the number of triangles formed by v_i.

Event: EventStream contans a sequence of Events, i.e., $\{e_1, e_2, \cdots, e_n\}$, where n is the number of events. We define an event for TC as (4):

$$e_i = (\,TYPE, m_i),\, 1 \leq \mathrm{i} \leq \mathrm{n} \tag{4}$$

where TYPE $\in \{ADD, UPDATE, DELETE\}$, and m_i is the event messages of e_i.

Transform: The state update process for TC algorithm is an instantiation of Algorithm 1. For a given event *e*, the source vertex v_{source} and the target vertex v_{target} can be got from GET-VALUE(*e*). For TC algorithm, there are two expanded functions: GET-NEIGHBOR(S_{source}) is used to get the neighbors of v_{source}, while GET-TRIANGLE(S_{source}) is used to get the triangle counter of v_{source}.

GTC is presented in Algorithm 1. A sequence of events (*EventStream*) is the input. For each event $e \in EventStream$, two adjacent sets are initialed first (lines 3-4): $adjacentSet_1$ stores the neighbor vertices of v_{source}, and $adjacentSet_2$ stores the neighbor vertices of v_{target}. After v_{source} and v_{target} are retrieved from *e* (line 5), based on FGL, these two vertices are locked to avoid potential update conflicts (lines 6-7). The next step (lines 9-19) is to compute the common adjacent neighbors of v_{source} and v_{target}. If f_{source} is not null, the set of adjacent neighbors $adjacentSet_1$ for v_{source} is the union of v_{target} and the previous neighbors of v_{source} (lines 11-12). Otherwise, the v_{target} would be the first neighbor of v_{target} (lines 13-14). The computation of $adjacentSet_2$ for v_{target} is similar (lines 15-18). We use *intersectSet* to store those common neighbor vertices (line 19). For each vertex in *intersectSet*, its triangle counts can be increased or decreased by one according to the event type (lines 20-27). Similarly, the factor of v_{source} and v_{target} can be updated based on event type (lines 28-37). When these update operations are finished, FGL will release the lock of v_{source} and v_{target}.

Fig. 1 shows an example of GraphFlow-based Triangle Count. In this example, suppose the latest graph state is as shown in Fig. 1(a). When an edge is added as shown in Fig. 1(b), we can calculate the common adjacent neighbors for v_{source} and v_{target} as shown in Fig. 1(c). Finally, the triangle counts of v_{source}, v_{target} and their common adjacent neighbors can be updated.

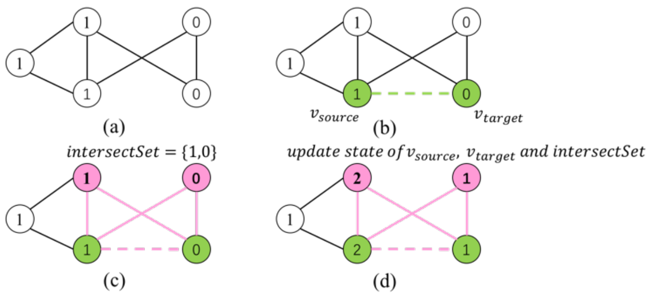

Fig. 1 GraphFlow-based Triangle Count Example.

Algorithm 1: GraphFlow-based Triangle Count (GTC)

1: **procedure** DTC(*EventStream*)
2:   **for all** $e \in EventStream$ **do**
3:     adjacentSet $_1 \leftarrow \emptyset$
4:     adjacentSet $_2 \leftarrow \emptyset$
5:     $(v_{source}, v_{target}) \leftarrow$ GET-VALUE(e)
6:     FGL_LOCK(v_{source})
7:     FGL_LOCK (v_{target})
8:     UPDATE(e)
9:     $f_{source} \leftarrow$ GET-STATE(v_{source})
10:     $f_{target} \leftarrow$ GET-STATE(v_{target})
11:     **if** $f_{source} \neq \emptyset$ **then**
12:       adjacentSet $_1 \leftarrow$ GET-NEIGHBOR(f_{source}) $\cup$ $\{v_{target}\}$
13:     **else**
14:       adjacentSet $_1 \leftarrow \{v_{target}\}$
15:     **if** $f_{target} \neq \emptyset$ **then**
16:       adjacentSet $_2 \leftarrow$ GET-NEIGHBOR(f_{target}) $\cup$ $\{v_{source}\}$
17:     **else**
18:       adjacentSet $_2 \leftarrow \{v_{source}\}$
19:     intersectSet $\leftarrow$ adjacentSet $_1 \cap$ adjacentSet2
20:     type $\leftarrow$ GET-TYPE(e)
21:     **for all** $v \in$ intersectSet **do**
22:       $f_v \leftarrow$ GET-STATE(v)
23:       t $\leftarrow$ GET-TRIANGLE(f_v)
24:       N $\leftarrow$ GET-NEIGHBOR(f_v)
25:       **if** type = ADD **then** $f_v \leftarrow$ (v, N, t+1)
26:       **elseif** type = DELETE **then** $f_v \leftarrow$ (v, N, t-1)
27:       SET-STATE(v, f_v)
28:     $t_{source} \leftarrow$ GET-TRIANGLE(f_{source})
29:     $t_{target} \leftarrow$ GET-TRIANGLE(f_{target})
30:     **if** type = ADD **then**
31:       $f_{source} \leftarrow (v_{source}$, adjacentSet $_1$, t_{source} + |intersectSet|)
32:       $f_{target} \leftarrow (v_{target}$, adjacentSet $_2$, t_{target} + |intersectSet|)
33:     **elseif** type = DELETE **then**
34:       $f_{source} \leftarrow (v_{source}$, adjacentSet $_1$, t_{source} - |intersectSet|)
35:       $f_{target} \leftarrow (v_{target}$, adjacentSet $_2$, t_{target} - |intersectSet|)
36:     SET-STATE(v_{source}, f_{source})
37:     SET-STATE(v_{target}, f_{target})
38:     FGL_UNLOCK(v_{source})
39:     FGL_UNLOCK (v_{target})