Maintaining Graph Properties over Sliding Windows

Rohit Kumar Toon Calders Aris Gionis
Nikolaj Tatti

DRAFT of June 18, 2015

1 Abstract 1

Large graphs are getting generated by applications which maintain a relationship between different data entities, for example: the social network in social media, publications and authors in DBLP, web pages and hyperlinks, sensor data, etc. Analyzing these graphs to compute different graph properties is an interesting problem area. The neighborhood density $N_G^x(t)$ of a graph gives the density of the Node x at distance t i.e the number of distinct nodes y from x at distance t. This could be used for finding many other interesting properties like the Neighborhood function $N_G(t)$[**?**] of a graph which gives many useful information of the graph like the diameter of the graph. Some of the best recent algorithms to calculate this properties are ANF[**?**] and HyperANF[**?**]. These algorithms are offline algorithms which does multiple passes of the given graph data and hence is not useful in case of streaming model of graph data. In streaming model the property of the graph changes and it would be interesting to calculate these properties at different time window to study the evolution of the graph. Hence in this work we present the first streaming algorithm to generate Neighborhood density of a graph on a sliding window model. The proposed algorithm is highly scalable as it permits parallel processing and hence can scale to very large graphs on a distributed system. We present results from both serial and parallel implementation of the algorithm for different social graphs.

2 Abstract 2

Large graphs are getting generated by applications which maintain a relationship between different data entities, for example: the social network in social media, publications and authors in DBLP, web pages and hyperlinks, sensor data, etc. There are many offline algorithms to compute approx or exact properties of these large graphs but all such algorithms assumes these graphs as static graph. Most real world graphs are dynamic where the edges are generated every second and with time the importance of an old edge decreases or diminishes. In this paper

we study the density profile of a graph, the r-density profile of the node v in graph G, $df^r_G(v)$, is the number of distinct nodes up-to distance r from node v. We present a streaming online algorithm to maintain the density profile of each node of a graph over a sliding window model. The proposed algorithm is highly scalable as it permits parallel processing and the computation is node centric, hence it can easily scale to very large graphs on a distributed system like Apache Giraph. We present results from both serial and parallel implementation of the algorithm for different social graphs.

3 Preliminaries

Let $G(V, E)$ be a graph. As usual, a path of length k between two nodes $v, w \in V$ in a graph $G(V, E)$ is a sequence of vertices $v_0 = v, v_1, \ldots, v_k = w$ such that for all $i = 1 \ldots k$, $\{v_{i-1}, v_i\} \in E$, and all v_i are different. The distance between v and w in the graph G is defined as the length of the shortest path between v and w if such a path exists, and otherwise is ∞. The distance between v and w in graph G will be denoted $d_G(v, w)$.

3.1 Evolving Graph Model

In this paper we will study dynamic graphs, that is, graphs in which the edges arrive over time. The same edge may appear multiple times in this stream of edges, denoting for instance repeated activity on the link between the two nodes. We assume the existence of a set of vertices V, and a stream $\langle e_1, e_2, e_3, \ldots \rangle$ of edges between the nodes in V; that is, $e_i \in V^{(2)} = \{\{u, v\} \mid u \neq v, u, v \in V\}$. For ease of notation, we assume that the ith edge "occurs" at time i. This implies that at time t we have seen the edges $\{e_i \mid i \in [1, t]\}$.

The edges may represent *activities* between the vertices, for instance:

1. V could be the set of employees of a company, and every edge represents communication (mail, phone call,...) between two employees.

2. V could be the users of a social network, and an activity between u and v occurs when one post something on the others wall

In our model we are only interested in recent events, and hence we assume a window length w is given. The *current window at time* t, denoted E_t consists of the edges $\{e_{t-w+1}, \ldots, e_t\}$, and is defined for all $t \geq w$. For every window time stamp t, the graph G_t is defined as the pair (V, E_t). We call G_t the *snapshot graph of the dynamic graph G at time t.*

3.2 Problem Statement

We are now ready to define the central notion in the paper, the density profile:

Definition 1 *Let $v \in V$ be a node. $N^i_G(v) = \{w \mid d_G(v, w) = i\}$, the* i-neighborhood of v in G, *denotes the set of all nodes that are at distance i of*

v. $n^i_G(v) = |N^i_G(v)|$ *denotes the cardinality of the* i*-neighborhood. Let* r *be an integer. We will call the sequence* $df^r_G = \langle n^1_G(v), n^2_G(v), \ldots, n^r_G(v)\rangle$ *the* r-density profile of the node v in graph G.

The problem that we are studying in the paper is how to maintain a summary of the streaming graph for the density profiles of all $v \in V$. We will slightly abuse notation and denote $n^i_{G(t)}(v), N^i_{G(t)}(v), df^i_{G(t)}(v)$ by respectively $n^i_t(v), N^i_t(v), df^i_t(v)$.

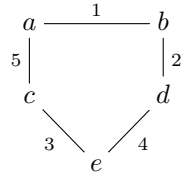
More concretely, we will propose a summary S_t of G_t with the following properties:

- The summary S_t of G_t requires limited storage space, and we can efficiently compute $n^i_t(v)$ from it. The time needed to compute $n^i_t(v)$ from S_t will be called *query time*.
- There is an efficient update procedure to compute S_{t+1} from S_t and e_{t+1}.

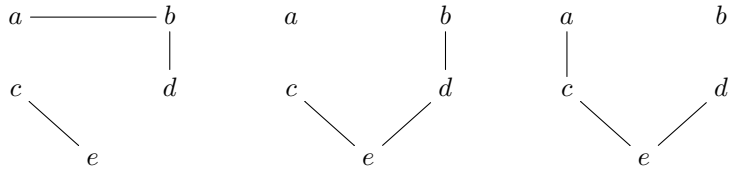
We will first introduce an exact, yet memory-inefficient exact solution. This exact solution will subsequently form the basis of a more memory-efficient approximate solution based on the well-known hyper loglog sketches. The approximate solution will not only be more memory-efficient, but also faster to compute.

(TOON) Possible target theory results: lower bound on exact solution.

Example 1 *Consider the following illustration of an edge stream over the set of nodes* $V = \{a, b, c, d, e\}$. *The numbers on the edges denote the time of arrival of the edges:*



Let the window length be 3. *Hence, at times* $t = 3, 4, 5$ *the snapshot graphs* G_t *are respectively:*



G_3 G_4 G_5

The 3*-density profiles of node* c *in these graphs are respectively* $(1, 0, 0)$, $(1, 1, 1)$, *and* $(2, 1, 0)$. *As can already be seen in this toy example, the density profile of subsequent windows can greatly differ.*

3.3 Maintaining the Neighborhood Distribution of all Nodes

We will now develop a method to maintain the density profiles of all nodes of the dynamic graph. As at time t wa are unaware of which nodes may arrive at later times, we define the notion of *possible futures* of a dynamic graph.

Definition 2 *Let $G_t = (V, E_t)$ be a snapshot of dynamic graph G at time t. Let $e_{w-t+1}, \ldots, e_t$ denote the sequence of edges of E_t in the order of the dynamic graph. The* possible futures of G_t after time span s *is the following set of graphs:*

$$\begin{aligned} \mathit{future}_s(G_t) \quad = \quad & \{G(V, \{e'_{t+s-w+1}, \ldots, e'_{t+s}\}) \mid \\ & e'_i = e_i \text{ for } i = t+s-w+1 \ldots \min\{t+s-w, t\} \\ & e'_i \in V^{(2)} \text{ for } i = \min\{t+s-w+1, t+1\}, \ldots, t+s\} \ . \end{aligned}$$

$future(G_t)$ denotes the union of all $future_s(G_t)$, $s = 0, 1, \ldots$

3.3.1 The summaries

Definitions still shaky ...

We first explain what the summary of G_t as a fixed time t looks like. The summary S_t will consist of the combination of all summaries S_t^v for $N_t^i G(v)$ of all nodes v and distances $i = 1 \ldots r$. An essential notion in the construction of the summary will be the that of a *promising path*. Let $e_1, e_2 \ldots$ denote the edges of a dynamic graph over V. All definitions in this section are relative to this given dynamic graph.

Definition 3 *Let $v, w \in V$ be two nodes that are connected by an edge in snapshot graph G_t. The* dawn of the edge $e = \{v, w\}$ in snapshot graph G_t, *denoted $dawn_t(e)$ is defined as* $\max\{i = t - w + 1 \ldots t \mid e_i = e\}$.

Let $p = \langle v_0, \ldots, v_k \rangle$ be a path in snapshot G_t. The dawn of the path p in snapshot G_t, *denoted $dawn_t(p)$, is the dawn of the oldest edge on that path; that is:*

$$\mathit{dawn}_t(p) \ := \ \min\{\mathit{dawn}_t(\{v_{i-1}, v_i\}) \mid i = 1, \ldots k\} \ .$$

(TOON) I replaced age by dawn as age is actually a very bad name as a higher age means a younger path ... I am not too happy with dawn either, though

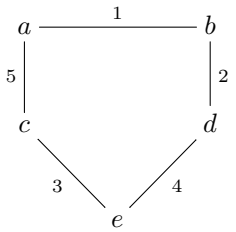
Definition 4 *Let $p = \langle v_0, \ldots, v_k \rangle$ be a path of length k in snapshot G_t. p is called* promising *if there exists a time span s and a graph $F \in future_s(G_t)$ such that p is a path in F, $dawn_t(p) = dawn_F(p)$, and $d_F(v_0, v_k) = k$. We will denote the set of all promising paths p between v and w in G_t by $Prom_t(v, w)$*

(Toon) *$dawn_F(p)$* **is not defined; needs rewriting. It becomes a mess as the edges in F are in principle unordered ...**

Proposition 1 *Let G_t be a snapshot. A path $p = \langle v_0, \rangle v_k$ of G_t is promising if and only if for all other paths p' between v_0 and v_k in G_t it holds that either p' is longer than p, or p' is older than p'; i.e., $dawn_t(p') \leq dawn_t(p)$.*

Proof. If p is a path of length at least 2, let e be the edge on the path p with the latest dawn. Otherwise e is any other edge in $V^{(2)}$. It is easy to see that the graph $F = (V, \{e_{dawn}(p), \ldots, e_t, e'_{t+1}, \ldots, e'_{dawn(p)+w-1} \mid e'_i = e \text{ for } i = t+1 \ldots dawn(p)+w-1\}$ is in $future(G_t)$, and p is a shortest path between v_0 and v_k in F. Indeed, the construction of F by adding only an already existing edge e is such that any path in F also exists in G_t. Furthermore, for any other path p' in G_t with $dawn_t(p') < dawn_t(p)$, it holds that p' is no longer a path in F because the oldest edge e' on the path p' in G_t has $dawn_t(e') = dawn_t(p') < dawn_t(p)$, and hence is not in F. □

Example 2 *We continue with the same example as before, but now with length equal to* 5*. The first complete snapshot* G_5 *of our dynamic graph hence looks as follows:*



The path $p_1 = \langle a, b, d\rangle$ *is a promising path of length* 2 *as it is the shortest path between* a *and* d *in the graph* G_t*. Its dawn is* 1*. Notice that this implies that the path* p_1 *between* a *and* d *will exist up to the moment that* e_1 *disappears from the sliding window. Furthermore, also the path* $p_2\langle a, c, e, d\rangle$ *is a promising path between* a *and* d*, even though it is not the shortest path between* a *and* d *in* G_t*. Indeed,* $dawn(p_2) = 3$*, and for any other path between* a *and* d *in* G_t *it holds that its dawn is earlier than* 3*. If now in the next step of the dynamic graph for instance the edge* $e_6 = \{a, c\}$ *would be inserted, the path* p_1 *disappears as* $e_1 = \{a, b\}$ *shifts out of the window and* p_2 *becomes the shortest path.*

Consider furthermore the paths between a *and* e*:* $p_3 = \langle a, c, e\rangle$ *and* $p_4 = \langle a, b, d, e\rangle$*.* p_3 *is a promising path, since it is already the shortest path between* a *and* e *in* G_t*.* p_4 *on the other hand is not a promising path since* p_4 *is longer than* p_3*, and older;* $dawn_t(p_3) = 1 < dawn_t(p_4) = 3$*. Hence,* p_4 *will never become the shortest path between* a *and* e*. This is indeed easy to see; since* $dawn_t(p_4) < dawn_t(p_3)$*, every future window that contains* p_4 *and has* $dawn(p_4) = 3$ *contains edge* e_3 *and hence will also contain* p_3*.*

Hence, it is clear that in order to be able to produce the density profiles of a node v in future, we need to maintain and keep track of the promising paths for which v is the starting node. As we will see, however, it is not necessary to maintain the complete list of all promising paths themselves; it is sufficient for every promising path to keep track of its length, dawn, and the nodes it connects. Therefore the summary we will be maintaining will be the combination, for every node v, of the following summaries $S_t^v := (S_t^v[1], \ldots, S_t^v[r])$ with

$$S_t^v[i] = \{(w, dawn_t(p)) \mid \exists p \in Prom_t(v, w) : len(p) = i\} .$$

Example 3 *We continue the example of Figure* **??**. *The window length w equals* 5*, and the maximal length up to which we need to track the density profile r is* 3*.*

For G_5, the first full window of the stream, the summary can schematically be represented as follows:

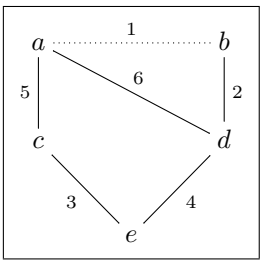
Node	*Summary*					
		to node				
	distance	a	b	c	d	e
a	1		1	5		
	2				1	3
	3				3	
b	1	1			2	
	2			1		2
	3			2		
c	1	5				3
	2		1		3	
	3		2			
d	1		2			4
	2	1		3		
	3	3				
e	1			3	4	
	2	3	2			
	3					

The summary S_t^a is: $(\{(b,1),(c,5)\},\{(d,1),(e,3)\},\{(d,3)\})$. $(d,1)$ in $S_t^a[2]$ denotes that there exists a promising path of length 2 *between a and d with dawn* 1*. Indeed, $\langle a,b,d\rangle$ is such a path.*

Update Rule

At time stamp $t+1$, S_t needs to be transformed into S_{t+1}. This implies that one edge is removed from the window (e_{t+1-w}), and one edge is added (e_t). Removing the edge e_{t+1-w} is easy enough; just prune any pair $(b, t+1-w)$ from $S_t^a[i]$ for all $a \in V$, $i = 1,\ldots,r$. This operation could also be postponed and executed in batch. Inserting the effects of the addition of the edge $e_t + 1$, however, is much more challenging. Let's first look at an example:

Example 4 *We continue the example of* **??**. *Suppose now that at time stamp* 6 *the edge between a and d is added, and the edge $e_1 = \{a,b\}$ is removed. The following graph is the resulting G_6 (the dotted edge e_1 is removed)*



The summary for G_6 becomes the following. The changes as compared to the summary of G_5 have been indicated in bold and outdated values have been strick through.

Node	*Summary*					
		to node				
	distance	a	b	c	d	e
a	1 2 3		~~1~~ **2**	5	**6** ~~1~~ ~~3~~	 ~~3~~**4**
b	1 2 3	~~1~~ **2**		 ~~1~~ 2	2	 2
c	1 2 3	5	 ~~1~~ 2		 ~~3~~**5**	3
d	1 2 3	**6**	2	 ~~3~~**5**		4
e	1 2 3	3	 ~~2~~**4**	3 **2**	4	

As can be seen in the example, the addition of an edge may result in a considerable number of changes in the summaries. Suppose an edge between nodes a and b is added at time $t+1$. Then, clearly the summaries of a and b need to be changed. Algorithm **??** depicts the procedure to propagate the changes incurred by the addition of an edge $\{a, b\}$. This addition is handled as a series of propagation whose handling may iteratively lead to other changes to be propagated. Cleverly combining the propagations may reduce the time needed to perform the necessary changes, although it won't change the theoretical complexity of the update algorithm.

3.3.2 The Algorithm

(Toon) Still needs to be cleaned w.r.t. notations, examples, etc. An adjacency list is maintained for all the nodes of the graph. When a new edge arrives between a and b then along with the node b the time stamp when the edge arrived is also stored in the adjacency list of a i.e $\aleph(a) = (b, t)$ and vice

verse for adjacency list of b. The over all summary S_a for a node a at distance d is a combination of distance summary sets, $D_r(a)$ $\forall r <= d$, where $D_r(a)$ is the set of node and time stamp pairs (v, t_v) such that till time stamp t_v there is a path of length r from a to v. When a new edge (a, b) arrives first the adjacency list of a and b is updated. If the edge between a and b has come for the first time then the new entry is added in the respective adjacency lists else if there was a previous edge between a and b then the previous entry is removed from the list and then the entry with new time stamp is added in the respective adjacency lists. The distance summaries $D_r(v)$ are updated as follows

1. With the addition of edge (a, b) D_1 of only a and b will change so $(b, t+1)$ is added to set $D_1(a)$, the addition (shown as $\oplus$ in the algorithm) checks if node b is already present in $D_1(a)$ and if present it updates the time stamp of b with $t + 1$ else it adds a new pair $(b, t + 1)$ in $D_1(a)$ similarly its done for $D_1(b)$.

2. All the neighbors of a are added into the propagation list for D_2 update as the new node b of a is now present at distance 2 from the neighbors of a. b will be also added in the propagation list for D_2 update as b is now neighbor of a and hence all the nodes which where at distance 1 from a are now distance 2 from b. Similarly all neighbors of b is also added in the propagation list.

3. From the propagation list the first item is picked and all the other items with the same node in the propagation list are processed and removed from the list.

4. After processing if the D_r of that node has changed then all its neighbors are added in the next propagation list for update of D_{r+1} because if some new node w has come at distance r for a node v then now w is at distance $r + 1$ for all the neighbors of v.

5. The update of the neighbors is described in the Algorithm **??**. If v is a node for which D_r has updated by addition of node w and u is the neighbor of v then the summary of $D_{r+1}(u)$ will be merged with summary $D_r(v)$. Now there could be two scenarios

   (a) w was not at distance $r + 1$ from u and in this case w will be added in the summary of $D_{r+1}(u)$ with the time stamp as minimum of the time between the edge u and v and between path v to w.

   (b) w was already at distance $r+1$ from u in this case only the appropriate time needs to be updated. as now there are two paths to w from u of length $r + 1$ one through v and one otherwise the maximum time of these two paths timestamps will be taken.

After merging the summary $D_r(u)$ is pruned and if any node is present in its smaller distance summaries for a latter timestamp then it is removed from $D_r(u)$.

Algorithm 1 AddEdge(edge $\{a, b\}$,time $t+1$)

Input: Summary $D_i(v)$ for all nodes $v \in V$ for all distance $i \in (1..d)$, time of the addition $t+1$, adjacency list $\aleph(v)$ for v=a,b.

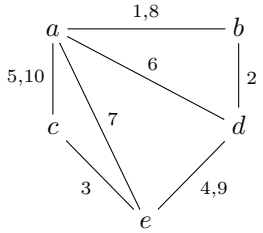
Output: Updated summaries $D_i(v)$ for all nodes $v \in V$ for all distances $i \in (1..d)$ and adjacency lists $\aleph(a)$ and $\aleph(b)$

```
if b ∈ ℵ(a) then
    remove b from ℵ(a)
end if
ℵ(a) = ℵ(a) + (b, t+1)
if a ∈ ℵ(b) then
    remove a from ℵ(b)
end if
ℵ(b) = ℵ(b) + (a, t+1)
P_next := {}
D_1(a) = D_1(a) ⊕ {b, t+1} and D_1(b) = D_1(b) ⊕ {b, t+1}
for all (v, t_v) ∈ ℵ(a) do
    P_next = P_next ∪ (v, 2, D_1(a), t_v)
end for
for all (v, t_v) ∈ ℵ(b) do
    P_next = P_next ∪ (v, 2, D_1(b), t_v)
end for
for r = 2..d do
    P_current := P_next
    P_next := {}
    while P_current ≠ {} do
        Pick (x, r, D_{r-1}(y), s) from P_current
        for all (x, r, D_{r-1}(y_x), s_x) ∈ P_current do
            Merge(x, r, D_{r-1}(y_x), s_x)
            Remove (x, r, D_{r-1}(y_x), s_x) from P_current
        end for
        if D_r(x) changes then
            add all neighbor of x from ℵ(x) in P_next
            for all (v, t_v) ∈ ℵ(x) do
                P_next = P_next ∪ (v, r+1, D_r(x), t_v)
            end for
        end if
    end while
end for
```

Algorithm 2 Merge$(x, r, D_{r-1}(y), t_{xy})$

Input: $D_r(x), D_{r-1}(y), t_{xy}$
Output: Updated summary $D_r(x)$
  for all $(v, t_v) \in D_{r-1}(y)$ **do**
    if $v \in D_r(x)$ for some t^* **then**
      $D_r(x) = D_r(x) \oplus (v, max(t^*, min(t_v, t_{xy})))$
    else
      $D_r(x) = D_r(x) \oplus (v, min(t_v, t_{xy}))$
    end if
  end for
  for all $(v, t_v) \in D_r(x)$ **do**
    if (v,t) is present in $D_i(x) \forall i < r$ and $t > t_v$ **then**
      remove (v, t_v) from $D_r(x)$
    end if
  end for

Example 5 *Assume the window length is 5, and that the edges arrive in the order indicated in the following illustration:*



Let's assume that we want to keep up to length 3.

(TOON) Redundant examples still need to be removed. We should have an example that walks the reader through the algorithm *After the*

first 5 steps the window is filled, and the summary is as follows:

Node	*Summary*					
		to node				
	distance	a	b	c	d	e
a	1		1	5		
	2				1	3
	3				3	
b	1	1			2	
	2			1		2
	3			2		
c	1	5				3
	2		1		3	
	3		2			
d	1		2			4
	2	1		3		
	3	3				
e	1			3	4	
	2	3	2			
	3					

Notice that the summary does not have $D_3(a) = (e, 1)$*, since there is already a more recent, shorter path to* e*, namely:* $D_2(a) = (e, 3)$*.*

At time stamp 6, the edge $\{a, d\}$ *arrives. As a result, the summaries are updated in a number of steps:*

1. *First the adjacency list of* a *and* d *is updated.* $\aleph(a) = \{(b, 1), (c, 5), (d, 6)\}$ *and* $\aleph(d) = \{(b, 2), (e, 4), (a, 6)\}$

2. *Then the* D_1 *for* a *and* d *is updated.*
   $D_1(a) = \{(b, 1), (c, 5), (d, 6)\}$
   $D_1(d) = \{(b, 2), (e, 4), (a, 6)\}$

3. *Now* D_2 *of* a *neighbors i.e* c*,* b *and* d *will be updated by merging it with* $D_1(a)$ *similarly* D_2 *of* a*,* b *and* e *will be merged by* $D_1(d)$*.*
   $D_2(c) = D_2(c)$ *merged* $D_1(a)$
   $=\{(b, 1), (d, 3)\} \oplus \{(b, max(1, min(1, 5))), (d, max(3, min(6, 5)))\}$
   $=\{(b, 1), (d, 5)\}$ *similarly,*
   $D_2(b) = \{(c, 1), (e, 2)\}$ *note that* $(d, 1)$ *is not added because* $(d, 2)$ *is already in* $D_1(b)$*.*
   $D_2(d) = \{(c, 5)\}$ *note* $(a, 1)$ *is removed as* $(a, 6)$ *is present in* $D_1(d)$*.*
   $D_2(a) = \{(e, 4), (b, 2)\}$ *note* $(d, 1)$ *is removed as* $(d, 6)$ *is present in* $D_1(a)$ *but* $(b, 2)$ *is added though* $(b, 1)$ *is present in* $D_1(a)$*.*
   $D_2(e) = \{(a, 4), (b, 2)\}$ *note time stamp for* a *is updated from* 3 *to* 4 *as there is a new path through* d *which is valid for higher time stamp.*

4. *Now as the summaries of only nodes* a*,* c*,* d *has changed* e *their neighbors will be updated for* D_3*.*

$D_3(a) = \{\}$
$D_3(b) = \{(c, 2)\}$
$D_3(c) = \{(b, 2), (e, 4)\}$
$D_3(d) = \{\}$
$D_3(e) = \{(c, 4)\}$

The algorithm will stop propagating at this time as we have reached the distance 3. So the summary would now look like the following

Node	*Summary*					
		to node				
	distance	a	b	c	d	e
a	1		1	5	6	
	2		2		1	4
	3					
b	1	1			2	
	2			1		2
	3			2		
c	1	5				3
	2		1		5	
	3		2			4
d	1		2			4
	2	6		5		
	3					
e	1			3	4	
	2	4	2			
	3			4		

The final summary after time stamp 10 would be

Node	*Summary*					
		to node				
	distance	*a*	*b*	*c*	*d*	*e*
a	1		8	10	6	7
	2				7	
	3					
b	1	8			2	
	2			8	6	7
	3				7	
c	1	10				3
	2		8		6	7
	3				7	
d	1	6	2			9
	2	7	6	6		
	3		7	7		
e	1	7		3	9	
	2		7	7		
	3					

3.4 Approximating the summaries

As we have seen though these summaries are very accurate in maintaining the Neighborhood Distribution of every node, the size of these summaries will be very large as the number of nodes at distance 3 or more will be very high. For highly dense graph the number of nodes at distance 2 itself will be large. We need to save these distance summaries for all nodes, one approach is to save these summaries in disk but that will result in slow processing and update time. If we want to keep these summaries in memory we have to make sure size of all the summaries is of the order of polylog(n) where n is number of edges. If we observer closely the summaries needs to have only two properties of a set operation.

1. They should have only distinct count of the nodes at a given distance. i.e A node u should be counted only once at distance x from v even if there are more than one path from v to u of length x.

2. The sets should be able to provide union operations.

We propose to use the HyperLogLog sketch for storing the distance summaries of every node. The HyperLogLog sketch is very memory efficient and meets both the criteria mentioned above. It requires only $loglog(n)$ bits of memory to keep a distinct count of n cardinality set and its is of $O(1)$ processing time per element. As we are looking for a sliding window model we will be using the sliding HyperLogLog sketch for every node. Sliding HLL is a bit more expensive than the normal HLL because in the sliding version we have to maintain list of

all possible future maximums (LPFM) for every bucket instead of just the maximum in the bucket as done in the HyperLogLog sketch. Even with this extra cost the sketch is still $O(log(n))$ for maintaining a distinct count n cardinality set with same precision.

3.4.1 Approximate counting in a window: Sliding-HyperLogLog sketch

HyperLogLog is an extension of the LogLog algorithm. Let M be the stream and h be a fixed suitable hash function. h divides the stream M into m sub streams $M_1, ...M_m$ using the first b bits of the hash where $m = 2^b$. An element in the stream after being hashed is divided into 2 parts. The first part(first b bits of the hash) is used to locate the register and from the remaining bits $\rho(u)$ is calculated as number of zero from left plus 1. The value at register $M[j]$ is given as :

$$M[j] = max(\rho(u)), u \in M_j$$

To calculate the distinct object seen so far the algorithm scans all the register $M[j]$ for $j = 1$ *to* m and calculates an indicator :

$$Z = \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$$

The cardinality E is then calculated as an harmonic mean of $2^{M[j]}$

$$E = \alpha_m m^2 Z$$

α_m is a constant to correct the systematic multiplicative bias present in $m^2 Z$ and is given as $\alpha_{16} = 0.673$; $\alpha_{32} = 0.697$; $\alpha_{64} = 0.709$; $\alpha_m = 0.7213/(1 + 1.079/m)$ for $m \geq 128$. The memory requirement for HyperLogLog is LogLogD for each register i.e O(mLogLogD) and the accuracy is given as $1.04/\sqrt{m}$.

In the Sliding-HLL sketch instead of storing just the maximum $\rho(u)$ in each bucket a list of all $\rho(u)$ and its arrival time stamp t_u is stored which could be a possible maximum in future when the current maximum is no more in the window. The list is called List of Future Possible Maxima (LFPM). One LFPM list is stored for each m bucket and updated independently. To estimate the number of distinct elements seen at time stamp t within a window of size w we extract from each LFPM the maximum $\rho(u)$ for which $t_u > t - w$. The harmonic mean and the estimate for number of distinct element is same as the HyperLogLog. Maintaining LFPM to store the possible future maximums has no impact on the accuracy of the estimate but the memory usage increase from $O(mloglog(n))$ to $O(mlog(n))$.

4 TO DO?

Some theory results still required:

- Space complexity of the complete sketch (almost done; is the aggregation of all loglog sketches)

- Batch updates; it seems logical that it doesn't matter in which order and how many updates at the same time are performed in the graph. If the updates are added to the propagation list and every time one is picked and we continue till the list is empty, it is logical that we will end up with a correct solution.

- Accuracy of the approximate solution. **(TOON) I conjecture that the only inaccuracy that is introduced is due to the hyerloglog sketch. In other words: if we run the exact solution and then transform the summary into hyperloglog the result will be exactly the same as the propagated version with the hyperloglog sketches. Proof (by induction, no doubt) seems straightforward but still needs to be made.**

- We need some results concerning the number of rounds needed to process an update. This depends on the order in which the propagations are performed. In a sequential setting it seems logical to process the propagation list nod by node, although even then a node needs to be updated multiple times. **(Toon) We need an example of that. Something with longer paths of later dawn causing propagations to arrive at different times in a node.** Another option is to select propagations based on their dawn order. It is however hard to predict what will be the effect of that as processing a propagation may cause new propagations of even earlier dawn (later not possible). In a distributed setting it is logical to process the propagation list in the order of distance; first we update all summaries at distance 1, then all at 2, then all at 3, etc. This allows for a higher level of parallelism and provably stops after r rounds.

- Some complexity analysis is still missing. Worst case (connecting two disconnected components of high degree nodes) is straightforward, but can we say something on average? **Nikolaj expressed some ideas concerning a paper that did something "on average" for pattern mining and that could be an inspiration ...**

- Should we Giraffe? It is so obvious to do Giraffe that it would be a shame to leave this for someone else ...