Maintaining sliding-window neighborhood profiles in interaction networks

Rohit Kumar[1], Toon Calders[1], Aristides Gionis[2], and Nikolaj Tatti[2]

[1]Department of Computer and Decision Engineering
Université Libre de Bruxelles, Belgium

[2]Helsinki Institute for Information Technology and
Department of Computer Science
Aalto University, Finland

Abstract. Large networks are being generated by applications that keep track of relationships between different data entities. Examples include online social networks recording interactions between individuals, sensor networks logging information exchanges between sensors, and more. There is a large body of literature on computing exact or approximate properties on large networks, although most methods assume static networks. On the other hand, in most modern real-world applications, networks are highly dynamic and continuous interactions along existing connections are generated. Furthermore, it is desirable to consider that old edges become less important, and their contribution to the current view of the network diminishes over time.

We study the problem of maintaining the neighborhood profile of each node in an *interaction network*. Maintaining such a profile has applications in modeling network evolution and monitoring the importance of the nodes of the network over time. We present an online streaming algorithm to maintain neighborhood profiles in the sliding-window model. The algorithm is highly scalable as it permits parallel processing and the computation is node centric, hence it scales easily to very large networks on a distributed system, like Apache Giraph. We present results from both serial and parallel implementations of the algorithm for different social networks. The summary of the graph is maintained such that query of any window length can be performed.

1 Introduction

Modern big-data systems are confronted with scenarios in which data are gathered in exceedingly large volumes. In many cases, the system entities are modeled as graphs, and the recorded data represent fine-grained activity among the graph entities. Traditionally, graph mining has focused on studying static graphs. However, as the emergence of new technologies makes it possible to gather detailed information about the behavior of the graph entities over time, a growing body of literature is devoted to the analysis of dynamic graphs.

In this paper we focus on a dynamic-graph model suitable for recording interactions between the graph entities over time. We refer to this model as *interaction networks* [26], while it is also known in the literature as *temporal networks* [21] or *temporal graphs* [23]. An interaction network is defined as a sequence of time-stamped interactions $\mathcal{E}$ over edges of a static graph $G = (V, E)$. In this way, many interactions may occur between two nodes at different time points. Interaction networks can be used to model the following modern application scenarios:

1. the set of nodes V represents the users of a social network or a communication network, and each interaction over an edge represents an interaction between two users, e.g., emailing, making a call, re-tweeting, etc.;
2. the set of nodes V represents autonomous agents, and each edge represents an interaction between two agents, e.g., exchanging data, being in the physical proximity of each other, etc.

We study the problem of maintaining the *neighborhood profile* of each node of a interaction network. In particular, we are interested in maintaining a data structure that allows to answer efficiently queries of the type *"how many nodes are within distance r from node v at time t?"* Graph neighborhood profiles have been studied extensively for static graphs [6, 25]. They provide a fundamental primitive for mining large graphs, either for characterizing the global graph structure, or for discovering important and central nodes in the graph. In this work, we extend the concept of neighborhood profiles for interaction networks, and we develop algorithms for computing neighborhood profiles efficiently in large and rapidly-evolving interaction networks. Our methods can be used for network monitoring, and allow detecting changes in the graph structure, as well as keeping track of the evolution of node centrality and importance.

To make our methods scalable to large and fast-evolving networks, we design our algorithms under the *data-stream model* [18, 24]. This model requires to process the interactions in an online fashion, and perform fast memory updates for each interaction processed. To make our model adaptable to changes and allow concept drifts we focus on the sliding-window model [14], a data-stream model that incorporates a forgetting mechanism, by considering, at any time point, only the most recent items up to that point. One uncommon benefit of our algorithm is that because of the data structure we incrementally maintain, the user can decide about the exact window length at query time.

Concretely, in this paper we make the following contributions: *(i)* we introduce a new problem of efficiently querying neighborhood profiles on interaction networks in Section 3; *(ii)* we develop and analyze an exact but memory-inefficient (Section 4) and an inexact but more efficient streaming algorithm for the sliding-window model (Section 5); *(iii)* we provide experimental validation of the algorithms in Section 7.

2 Preliminaries

We consider a static underlying graph $G = (V, E)$. An *interaction* over G is a time-stamped edge $(\{v, w\}, t)$ indicating an interaction between nodes v and w.

An *interaction network over* G is now defined as a pair $(G, \mathcal{E})$, where G is a static graph and $\mathcal{E}$ is a set of interactions. We should point out that we do not need to know E beforehand.

If the set of interactions $\mathcal{E} = \{(\{u, v\}, t)\}$ is ordered by time, it can be seen as a *stream of edges*, and written as $\mathcal{E} = \langle (e_1, t_1), (e_2, t_2), \ldots \rangle$, with $t_1 \leq t_2 \leq \ldots$. Note that two fixed nodes may interact multiple times in $\mathcal{E}$.

In our model we are only interested in recent events, and hence queries over our interaction network will always include a window length w — recall that the summary will be maintained in such a way that all window lengths are possible, i.e., every query can use a different window length. The *snapshot graph at time* t *for window* w, denoted $G(t, w)$, is the triplet $(V, E(t, w), \mathit{recent})$ in which $E(t, w) = \{e \mid (e, t') \in \mathcal{E} \text{ with } t - w < t' \leq t\}$, and *recent* is a function mapping an edge $e \in E(t, w)$ to the most recent time stamp that an interaction between the endpoints of e occurred, that is, $\mathit{recent}(e) = \max\{t' \mid (e, t') \in \mathcal{E} \text{ such that } t - w < t' \leq t\}$.

Furthermore, for the graph G we have the usual definitions; a *path* of length k between two nodes $u, v \in V$ is a sequence of nodes $u = w_0, \ldots, w_k = v$ such that $\{w_{i-1}, w_i\} \in E$, for all $i = 1, \ldots, k$, and all w_i are different. The *distance* between u and v in the graph G is defined as the length of the shortest path between u and v, if such a path exists, otherwise it is infinity. The *distance* between nodes u and v in the graph G is denoted by $d_G(v, w)$, or simply $d(v, w)$, if G is known from the context.

3 Problem statement

The central notion we are computing in this paper is the *neighborhood profile*:

Definition 1. *Let $G = (V, E)$ be a graph and let $u \in V$ be a node. The* r-neighborhood *of u in G, denoted $N_G(u, r)$, is is the set of all nodes that are at distance r from node u, i.e., $N_G(u, r) = \{v \mid d_G(u, v) = r\}$. We write $n_G(u, r) = |N_G(u, r)|$ to denote the cardinality of the r-neighborhood. We will call the sequence $p_G(u, r) = \langle n_G(u, 1), n_G(u, 2), \ldots, n_G(u, r) \rangle$ the* r-neighborhood profile *of the node u in graph G.*

In this paper we study the problem of *maintaining* the neighborhood profile $p_{G(t,w)}(u, r)$, for all nodes $u \in V$, as new interactions arrive in $\mathcal{E}$. Our solution allows w to vary; hence, at a time point t, we should be able to query for the neighborhood profile $p_{G(t,w)}(u, r)$ for *any* w. If there is an upper bound given for w, say w_{max}, then we can use this information to improve memory consumption. However, this is optional, and we can set $w_{max} = \infty$. On the other hand, r is given and fixed. Obviously, by computing $p_{G(t,w)}(u, r)$ we also compute $p_{G(t,w)}(u, r')$ for $r' < r$.

Let $H = G(t, w)$. To simplify the notation we will denote $N_H(u, r)$, $n_H(u, r)$, $p_H(u, r)$ by $N_{t,w}(u, r)$, $n_{t,w}(u, r)$, $p_{t,w}(u, r)$, respectively. Moreover, if $w = w_{max}$, then we will use $N_t(u, r)$, $n_t(u, r)$, $p_t(u, r)$, respectively. We will also write $G(t) = G(t, w_{max})$ and $E(t) = E(t, w_{max})$.

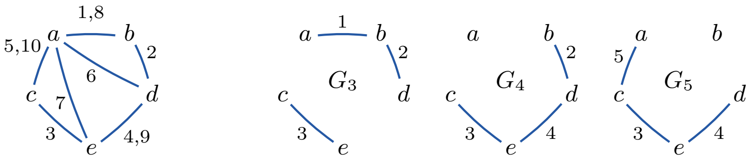


Fig. 1. A toy interaction network, and three snapshot graphs with a window size of 3.

Example 1. Consider the illustration given in Figure 1 of an edge stream over the set of nodes $V = \{a, b, c, d, e\}$. The numbers on the edges denote the time of interactions over the edges. Let the window length be 3. The snapshot graphs $G(t)$ at times $t = 3, 4, 5$ are also depicted in Figure 1. The 3-neighborhood profiles of node c in these graphs are respectively $(1, 0, 0)$, $(1, 1, 1)$, and $(2, 1, 0)$.

To accomplish our goal we maintain a summary S_t of the snapshot graph $G(t, w_{max})$, from which we can efficiently compute the neighborhood profiles $p_{t,w}(u, r)$, for every node u in the graph G. More concretely, we require that the summary S_t has the following properties:

1. The summary S_t of $G(t, w_{max})$ should require limited storage space.
2. The size of the r-neighborhood $n_{t,w}(u, r)$ should be easy to compute from S_t. The time to compute $n_{t,w}(u, r)$ from S_t will be called *query time*.
3. There should be an efficient update procedure to compute S_{t_i} from $S_{t_{i-1}}$ and the edge e_{t_i} on which the interaction at time-stamp t_i is taking place.

4 Maintaining the exact neighborhood profile

We first introduce an *exact*, yet memory-inefficient solution. This exact solution will form the basis of a memory-efficient and faster *approximate* solution based on the well-known *hyperloglog sketches*.

4.1 Summary for neighborhood functions

An essential notion in our solution is the *horizon of a path*, which expresses the latest time that needs to be included in the sliding window in order for the path to exist; i.e., if the sliding window starts after the horizon the path will not exist in it anymore.

Definition 2. *Let $G(t) = (V, E, recent)$ be a snapshot graph and $p = \langle v_0, \ldots, v_k \rangle$ a path in it. The* edge horizon *of p in $G(t)$, denoted by $h_t(p)$, is the time stamp of the* oldest *edge on that path: $h_t(p) = \min\{recent((v_{i-1}, v_i)) \mid i = 1, \ldots, k\}$.*

We will next define the horizon between two nodes u and v. Let $\mathcal{P}_H(u, v)$ be all the paths from u to v in a graph H. If $H = G(t)$, then we will write $\mathcal{P}_t(u, v)$.

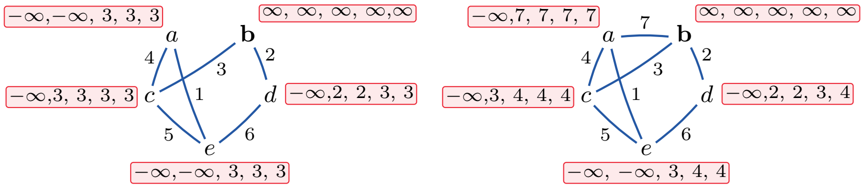


Fig. 2. Two toy snapshot graphs along with $h(u, b, i)$ for $i = 0, \ldots, 4$.

Definition 3. *The horizon for length i between two different nodes u and v is the maximum horizon of any path of at most length i between them; that is, $h_t(u, v, i) = \max\{h_t(p) \mid p \in \mathcal{P}_t(u, v), |p| + 1 \leq i\}$. We set $h_t(u, v, i) = -\infty$ if no such path exists. For any node u, $h_t(u, u, i)$ is defined to be ∞.*

Example 2. Consider the leftmost graph given in Figure 2, along with, for every node $u \in \{a, b, c, d, e\}$, the list of horizons $h(u, b, 0), \ldots, h(u, b, 4)$. In this graph $h(d, b, 1) = h(d, b, 2) = 2$, as there is an edge with a time stamp of 2. However, $h(d, b, 3) = 3$ as there is a path $\langle d, e, c, b\rangle$ with a horizon of 3.

The horizon between two nodes u and v for a length i is very important for our algorithm as it expresses in which windows u and v are at a distance i or less. Windows that include the horizon will have the nodes at distance i, shorter windows will not. Hence, if for a node u we know all horizons $h_t(u, v, i)$, for all distances i and all other nodes v, we can give the complete neighborhood profile for u for any window length. Hence, the summary S_t of the snapshot graph $G(t)$ will be the combination, for all nodes u and distances $i = 0, \ldots, r$, of the summaries S_t^u for $N_t(u, i)$. In other words, for every node u, we will be maintaining the summary $S_t^u = (S_t^u[0], \ldots, S_t^u[r])$, where $S_t^u[i] = \{(v, h_t(u, v, i)) \mid h_t(u, v, i) > -\infty\}$.

Example 3. For the snapshot graph given in Fig. 2, the summary S_t consists of $S^u[i]$, $i = 0, \ldots, r$. Assuming $r = 3$, the summaries for a and b are as follows:

S^a					
distance	a	b	c	d	e
0	∞				
1	∞	3	4		1
2	∞	3	4	1	4
3	∞	3	4	4	4

S^b					
distance	a	b	c	d	e
0		∞			
1		∞	3	2	
2	3	∞	3	2	3
3	3	∞	3	3	3

4.2 Updating summaries

We describe how to update the summary S_t as new edges arrive in the stream $\mathcal{E}$ or old edges expire. The latter event happens for edges whose time-stamp

Algorithm 1: ADDEDGE($\{a, b\}, t$), updates a summary upon addition of $\{a, b\}$ at time t

```
foreach $i = 0, \ldots, r - 1$ and $(x, t') \in S^a[i]$ do  $g(b, x, i + 1) \leftarrow \min(t', t)$ ;
foreach $i = 0, \ldots, r - 1$ and $(x, t') \in S^b[i]$ do  $g(a, x, i + 1) \leftarrow \min(t', t)$ ;
PROPAGATE($\{g(v)\}_{v \in V}$)
```

Algorithm 2: PROPAGATE($\{g(v)\}_{v \in V}$), Processes all propagations that are in the general register g.

```
foreach $i = 1, \ldots, r$ do
    foreach $v, x \in V$ such that $g(v, x, i)$ is set do
        if MERGE($x, v, g(v, x, i), i$) then
            foreach $(v, u) \in E_t \setminus \{a, b\}$ do
                $horizon \leftarrow \min(g(u, x, i), recent(v, u))$;
                if $g(u, x, i + 1)$ not set or $horizon > g(u, x, i + 1)$ then
                    $g(u, x, i + 1) \leftarrow horizon$;
```

becomes smaller than $t - w_{max}$. Removing an edge is easy enough; we need to remove all pairs (x, t') from summaries $S_t^u[i]$, for all $u, x \in V$, $i = 1, \ldots, r$, and $t' \leq t - w_{max}$. This operation could also be postponed and executed in batch. Updating the summary S_t to reflect the addition of a new-coming edge e_t, however, is much more challenging. Let us first look at an example.

Example 4. Consider the horizons of the two graphs given in Figure 2. Notice that adding an edge $\{a, b\}$ changed $h(d, b, 4)$ from 3 to 4 because we introduced a path $\langle d, e, c, a, b \rangle$. However, the key observation is that we also changed $h(e, b, 3)$ to 4 due to the path $\langle e, c, a, b \rangle$, $h(c, b, 2)$ to 4 due to the path $\langle c, a, b \rangle$, and $h(a, b, 1)$ to 6 due to the path $\langle a, b \rangle$.

As can be seen in the example, the addition of an edge may result in a considerable number of non-trivial changes. However, the example also hints that we can propagate the summary updates.

Assume that we are adding an edge $\{a, b\}$, and this results in change of $h(u, v, i)$. This change is only possible if there is a path $p = \langle u = v_0, \ldots, v_k = v \rangle$ through $\{a, b\}$. Moreover, we will also change $h(u, v_{k-1}, i - 1)$. By continuing in this logic, it is easy to see that all the updates can be processed via a *breadth-first search* from node b. Furthermore, whenever we can conclude that $h(u, v, i)$ does not need to be updated, we can stop exploring this branch since we know that no extensions of this path will result in updates. The pseudo-code for this procedure is given in Algorithms 1–3.

In the algorithm we update the summaries, distance by distance, and we set new (earlier) horizons that have possibly appeared due to the newly added edge. To maintain the updates we use a function g; $g(u, x, i) = h$ indicates that there is

Algorithm 3: Merge(x, v, t, i), adds x to a summary of v with a distance of i and edge horizon t. If false is returned, then the branch can be pruned.

```
if $(x, t') \in S^v[i]$ for some $t' \geq t$ then return false;
remove all $(x, t')$ from $S^v[i]$ for which $t' < t$;
add $x, t$ to $S^v[i]$;
return true;
```

a new path between u and x of length i and horizon h. As not every new path of length i will lead to an improved horizon, we do not propagate this information immediately to the summary of the neighboring nodes, but rather wait until we have processed all paths of length $i - 1$. For those new paths that improve the summary of a node u, we will then propagate this information further on in the graph. For every distance i, when we process an update to a summary we will record potential updates to horizons of length $i + 1$ as follows: if $g(u, x, i)$ leads to a better horizon of length i between u and x; that is, either there is not yet an entry (x, h) in $S^u[i]$, or $h < g(u, x, i)$, then we will propagate this information to its neighbors u. Let $t = \min(\mathit{recent}(u, v), g(v, x, i))$, then we will propagate $g(u, x, i + 1) = t$, if $t > g(u, x, i + 1)$, that is, we were able to improve our potential update.

Example 5. We will continue our running example given in Figure 2. Let us demonstrate how the horizons of $h(u, b, i), u \in \{a, b, c, d, e\}$ are updated once we introduce the edge $\{a, b\}$. In Figure 3 we illustrate how the propagation is done. At the beginning of each round we compare the current summary $S^u[i]$ against the new candidate horizon $g(u, b, i)$. If the latter is larger, then we update the summary as well as propagate new candidate horizons to the neighboring nodes. In the subsequent figures it is indicated what are the changes with respect to the distances to node b. In the first step, due to the addition of edge $\{a, b\}$ at time 7, for distance 1 the update $g(a, b, 1) = 7$ is propagated. When processing this update indeed it is seen that the summary $S^a[1]$ is updated. Therefore, this update is further propagated to the neighbors, leading to the following

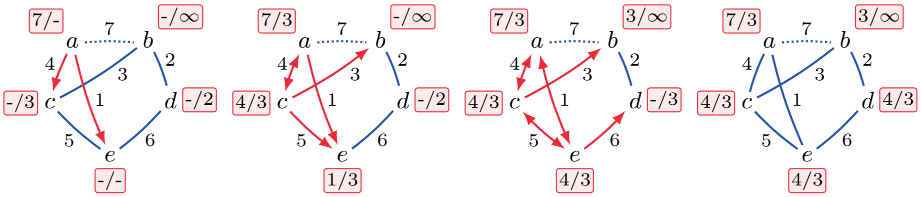


Fig. 3. Propagation of updates for the vertex b when adding (a, b) for the rounds $i = 1, \ldots, 4$. The format of boxes is y/z, where y is the time of b in $S^v[i]$ and $z = g(v, b, i)$ at the beginning of ith round. The edges used for propagation during ith round are marked in red. We do not show propagation during the last round as it is not needed.

updates: $\{g(c,b,2)=4, g(e,b,2)=1\}$. As only the first update changes the summary $S^c[2]$, only this update will be further propagated. Furthermore, for a there is the update $g(a,b,2)=7$ that needs to be processed. Propagation leads to the following new updates (first three for $g(c,b,2)$, last two for $g(a,b,2)$): $\{g(a,b,3)=4, g(b,b,3)=3, g(e,b,3)=4, g(c,b,3)=4, g(e,b,3)=1\}$. The last update $g(e,b,3)=1$ will never be considered as it is dominated by the update $g(d,b,3)=4$. These updates are then processed and those implying changes in the summary are again propagated.

The proofs of the following proposition is omitted due to space constraints.

Proposition 1. AddEdge *updates the summary correctly. Let $n=|V|$, $m=|E|$, and r be the upper bound on the distances we are maintaining. The time complexity of* AddEdge *is $\mathcal{O}(rmn\log(n))$. The space complexity is $\mathcal{O}(rn^2)$.*

5 Approximating neighborhood function

The algorithm presented in the previous section computes the neighborhood profiles exactly, albeit, it has high space complexity and update time. In this section we describe an approximate algorithm, which is much more efficient in terms of memory requirement and update time.

The approximate algorithm is based on an adaptation of the hyperloglog sketch [17] to the sliding-window context, similar to the adaptation by Chabchoub and Hébrail [9]. The resulting sliding hyperloglog sketch has the following properties: (i) it provides a compact summary of a stream of items, and (ii) it allows to answer the following question: *"How many different items have appeared in the stream since a given time point t?"* Subsequently, this sketch can replace the neighbor sets that need to be maintained by the exact algorithm.

5.1 Hyperloglog and sliding-window hyperloglog sketches

The hyperloglog sketch [17] consists of an array of numbers, whose size is 2^k, and a hash function η that assigns each item of the stream in a uniformly-random number in the range $[0, 2^n-1]$. The value of n should be sufficiently large in the sense that 2^{n-k} should significantly exceed M, the number of distinct items in the stream. We will use the standard assumption that $n \in \mathcal{O}(\log M)$. Initially all cells of the hyperloglog sketch are set to 0. The update procedure for the hyperloglog sketch is as follows: if an item x arrives in the stream, the first k bits of the binary representation of $\eta(x)$ are used to determine which entry of the sketch array will be updated. We denote this index by $\iota(x)$. From the remaining $n-k$ bits $\eta'(x)$, the quantity $\rho(x)$ is computed as the number of trailing bits in the binary representation of $\eta'(x)$ that are equal to 0, plus 1. If the current value at the entry $\iota(x)$ of the sketch is smaller than $\rho(x)$, we update the value of that entry. Clearly, the more different items in the stream, the more likely it is to observe large tails of 0's and the higher the numbers in the hyperloglog sketch will become.

In order to make the hyperloglog sketch working in the sliding-window setting, we need to store multiple values per entry. Initially the sliding-HLL sketch will start with an *empty set* for each entry. The process a new item x arriving in the stream at time t, we first need to retrieve the set of time-value pairs associated with the index $\iota(x)$. We then need to add the pair $(t, \rho(x))$ to that set and remove all entries (t', β) for which $\beta \leq \rho(x)$ (as t is the most recent time-stamp, it is also $t' < t$). We denote the sliding-HLL sketch after processing the stream of events $\mathcal{S} = \langle \sigma_1, \ldots, \sigma_n \rangle$ by *sHLL*$(\mathcal{S})$. More formally:

Definition 4. *Let $S = \{(t_1, \beta_1), \ldots, (t_n, \beta_n)\}$ be a set of time-value pairs. Define the subset of time-decreasing values of S as*

$$dec(S) = \{(t_i, \beta_i) \mid \beta_i > \beta_j \text{ for all } (t_j, \beta_j) \in S \text{ with } t_i \leq t_j\}.$$

A sliding hyperloglog sketch sHLL of dimension k is an array of length 2^k in which every entry contains a set of time-value pairs. For a stream $\mathcal{S}$, sHLL$(\mathcal{S})$ is recursively defined as follows:

- *If $\mathcal{S} = \langle\rangle$, then $sHLL(\mathcal{S})[i] = \{\}$, for all indices $i = 1 \ldots 2^k$.*
- *Otherwise, if $\mathcal{S} = \langle \mathcal{S}', (x, t) \rangle$ then $sHLL(\mathcal{S})[i] = dec(sHLL(\mathcal{S}')[i] \cup \{(t, \rho(x)\})$ for $i = \iota(x)$; while $sHLL(\mathcal{S})[i] = sHLL(\mathcal{S}')[i]$ for all other $i = 1 \ldots 2^k$.*

Example 6. Suppose that the hash η, ι, and ρ are as follows (recall that η determines the other two quantities):

item	a	b	c	d	e
η	100 01	101 11	010 11	010 10	001 10
ι	1	3	3	2	2
ρ	3	1	2	2	1

For the stream of items a, b, a, c, d, e, the resulting sliding HLL sketches are respectively the following:

$$\begin{array}{c|c|c|c|c|} \iota & 0 & 1 & 2 & 3 \\ \hline \rho & \{\} & \{\} & \{\} & \{\} \\ \hline \end{array} \xrightarrow{a} \begin{array}{c|c|c|c|c|} \iota & 0 & 1 & 2 & 3 \\ \hline \rho & \{\} & (1,3) & \{\} & \{\} \\ \hline \end{array}$$

$$\xrightarrow{b} \begin{array}{c|c|c|c|c|} \iota & 0 & 1 & 2 & 3 \\ \hline \rho & \{\} & (1,3) & \{\} & (2,1) \\ \hline \end{array} \xrightarrow{a} \begin{array}{c|c|c|c|c|} \iota & 0 & 1 & 2 & 3 \\ \hline \rho & \{\} & (3,3) & \{\} & (2,1) \\ \hline \end{array}$$

$$\xrightarrow{c} \begin{array}{c|c|c|c|c|} \iota & 0 & 1 & 2 & 3 \\ \hline \rho & \{\} & (3,3) & \{\} & (4,2) \\ \hline \end{array} \xrightarrow{d} \begin{array}{c|c|c|c|c|} \iota & 0 & 1 & 2 & 3 \\ \hline \rho & \{\} & (3,3) & (5,2) & (4,2) \\ \hline \end{array}$$

$$\xrightarrow{e} \begin{array}{c|c|c|c|c|} \iota & 0 & 1 & 2 & 3 \\ \hline \rho & \{\} & (3,3),(6,1) & (5,2) & (4,2) \\ \hline \end{array}$$

When b arrives, cell 3 gets value 1, which is updated later on when c arrives, since c has the same index, but a higher value. For d and e the situation is opposite; first d arrives giving a value of 2 in cell 2. Later on, when e arrives this value is not updated even though e has the same index because its value is lower.

The next proposition shows that with the sliding HLL sketch we can indeed obtain an approximate answer regarding the number of different items since time s, for any s specified at query time. We omit the proof as it follows immediately from the definition.

Proposition 2. *Let $\mathcal{S} = \langle \sigma_1, \ldots, \sigma_n \rangle$ be a stream of events in which event σ_t arrives at time t. Then for every index $1 \leq s \leq n$, it holds that for every entry $i = 1, \ldots, 2^k$, it is $\mathit{HLL}(\sigma_s, \ldots, \sigma_n)[i] = \max\{r \mid (t, r) \in \mathit{sHLL}(\mathcal{S})[i]$ and $t \geq s\}$, where* $\max(\{\}) = 0$.

5.2 Computation of neighborhood profiles based on sliding HLL

We are now ready to describe our technique for computing the approximate neighborhood profiles. Recall that we are working over a streaming graph with nodes from a set V and a stream of edges $\mathcal{E} = \{(e_1, t_1), (e_2, t_2), \ldots\}$. We have used E_t to denote the set of edges arrived until time t, i.e., $E_t = \{(e, t') \in \mathcal{E} \mid t' \leq t\}$. The approximate sketch is very similar to the exact sketch, with the exception that all sets of (node,time)-pairs are replaced by the much more compact sliding HLL sketch. Furthermore, in order to be able to propagate the updates to its neighbors, for every node we should know its neighbors. Hence, at time t, the summary consists, for every node u, of the following components:

$$N_t^u = \{(v, \mathit{recent}(u, v)) \mid (u, v) \in E_t\} \quad \text{and} \quad C_t^u = \langle C_t^u[1], C_t^u[2], \ldots, C_t^u[r] \rangle,$$

where $C_t^u[i] = \mathit{sHLL}(\{(v, h_t(p)) \mid p \in \mathcal{P}_t(u, v), |p| \leq i\})$.

The set N_t^u specifies the neighbors of node u in the graph $G_t = (V, E_t)$. Note that in the set N_t^u we keep pairs (v, t) such that v is a neighbor of u and t is the most recent time-stamp that an interaction between u and v took place. This time-stamp is needed to decide whether the neighbor v is active for a given window length that is specified at query time.

To update the summary C_t from the summary at the previous time instance, after the addition of an edge (a, b) at time t, we follow the almost exact same propagation method as the exact algorithm. The only difference is that instead of keeping all pairs $(v, h_t(p))$, we now keep a sliding HLL sketch over those pairs, as specified in the previous section. Updating a sliding HLL sketch is slightly more involved than updating the exact summary since we need to keep the sketch as a time-decreasing sequence. The pseudo-code for this is given in Algorithm 4.

Finally, to update the sketch, we use Algorithms 1 and 2, with the exception that the summary $S^u[\cdot]$ is replaced with the sketch $C^u[\cdot][j]$ for a fixed bucket j. We then execute 2^k copies of the algorithm, each handling its own bucket. As these algorithms are syntactically the same to the ones of the exact algorithm, we omit them. The proof of the following proposition is omitted due to space constraints.

Proposition 3. *The sketch version of* AddEdge *performs correctly. Let $n = |V|$, $m = |E|$, and r be the upper bound on the distances we are maintaining. The time complexity of the sketch version of* AddEdge *is $\mathcal{O}(2^k rm \log^2(n))$. The space complexity is $\mathcal{O}(2^k nr \log^2 n)$.*

Note that a naïve way to maintain approximate neighborhood profiles is to execute the sketching algorithm from scratch after each newly-arriving interaction. In the worst case, this brute-force method has roughly the same space and

Algorithm 4: SketchMerge(x, v, t, i), adds x to a summary of v with a distance of i and edge horizon t.

```
if $(y, t') \in C^v[i]$ for some $t' \geq t$, $y \geq x$ then return false;
remove all $(y, t')$ from $C^v[i]$ for which $t' \leq t$ and $y \leq x$;
add $(x, t)$ to $C^v[i]$;
return true;
```

time complexity as our incremental algorithm. However, the brute-force method is expected to require as much space and time as indicated by the worst-case bound, while for our method the worst-case analysis is very pessimistic: most of the times the summaries will not by propagated at the whole network and updates will be very fast. This is demonstrated in our experimental evaluation.

6 Related work

During the last two decades, a large body of work has been devoted to developing algorithms for mining data streams. Interestingly, the area started with processing *graph streams* [20], but a lot of emphasis was put on computing statistics over streams of items [12,18], and many fundamental techniques have been developed for that setting. Many different models have been studied in the context of data-stream algorithms, including the *sliding-window* model [14], which incorporates a forgetting mechanism where data items expires after W time units from the moment they occur. Existing work has considered estimating various statistics in this model [2,3].

The concept of *sketching* is closely related to data streams, as efficient streaming algorithms operate by maintaining compact sketches, which provide approximate statistics and summaries of the data stream seen so far. Popular data-stream sketches include the *min-hash sketch* [10], the *LogLog sketch* [15], and its improvement, the *hyperloglog sketch* [17], all of which have been used to approximate distinct counts. *Distance distribution sketches* [6,11] are built on top of the distinct-count sketches, and provide a powerful technique to approximate the number of neighbors of a node in a graph within a certain distance. Such sketches have been used extensively in graph-mining applications [6,25].

As graphs provide a powerful abstraction to model a wide variety of real-world datasets, and as the amount of data collected gives rise to massive graphs, there is growing interest on algorithms for processing *dynamic graphs* and *graph streams*. This includes work on data structures that allow to perform efficient queries under structural changes of the graph [16,19], as well as the design of algorithms for computing graph primitives under data-stream models. Work in the latest category includes algorithms for counting triangles [4,5,27] and other motifs [7,8], computing graph sparsifiers [1], and so on. Most of the above papers consider the standard data stream model, although Crouch et al. [13] study many graph algorithms on the sliding-window model.

Table 1. Characteristics of interaction networks.

Dataset	Nodes	Distinct edges	Total edges	Clustering coefficient	Diameter	Effective diameter
Facebook	4 039	88 234	88 234	0.60	8	4.7
Cit-HepTh	27 771	352 801	352 801	0.31	13	5.3
Higgs	166 840	249 030	500 000	0.19	10	4.7
DBLP	192 357	400 000	800 000	0.63	21	8.0

Table 2. Average relative error as a function of ℓ.

ℓ	Facebook	Cit-HepTh	Higgs	DBLP
16	0.28	0.23	0.22	0.22
32	0.13	0.16	0.19	0.15
64	0.10	0.12	0.16	0.12
128	0.08	0.10	0.14	0.09

7 Experimental evaluation

We provide an empirical evaluation of the approximate algorithm presented in Section 5. We evaluate the space requirements, time, and accuracy. We compare the approximate algorithm with the exact algorithm presented in Section 4 and the *off-line* HyperANF algorithm [6]. Since our implementations have not been optimized, we compare to a HyperANF version developed under the same conditions and without low-level optimizations such as broad-word computing.

Datasets and setup: We use four real-world datasets obtained from SNAP repository [22]. We take snapshots of the largest datasets Cit-HepTh and DBLP of 500 000 and 400 000 edges, respectively. Three of the data sets, Facebook, DBLP, and Cit-HepTh, have unique edges and do not contain any time information. To create an interaction network out of these static graphs, we order the edges randomly. In the case of DBLP we allow edges to repeat until we have 800 000 edges. Statistics of these datasets are reported in Table 1.

As a maximum window size we use $w_{max} = \infty$, that is, we do not delete any previous edges. We also set $r = 3$, except for one experiment where we vary r.

Accuracy of the sketch: In order to test the accuracy of the sketch algorithm, we compare the algorithm with the exact version, and we compute the average relative error as a function of number of buckets ($\ell = 2^k$). Running the exact algorithm is infeasible for the large datasets due the memory requirements, and hence we use only a subset of the large datasets to measure accuracy. The results are given in Table 2. As expected from previous studies, the accuracy increases with ℓ.

Running time for updating summaries: Our next goal is to study the running time needed to update the summary upon adding an edge. The average

Table 3. Average time in seconds needed to process 1 000 edges as a function of ℓ

ℓ	Facebook	Cit-HepTh	Higgs	DBLP
16	0.06	7.20	3.92	0.80
32	0.08	12.57	6.84	1.31
64	0.12	28.64	12.12	2.10
128	0.17	50.74	21.38	3.45

running time for every 1 000 edges is reported in Table 3.[1] Detailed time measurements are shown in Figure 4. We took average run time by running 3 iteration of `Facebook` and `Cit-HepTh` and 2 iteration of `Higgs` and `DBLP` datasets.

The time needed to process an edge depends on two factors. First, as we increase the number of buckets ℓ, the processing time increases. Second, a single edge may cause a significant number of updates if it connects two previously disconnected components. We see the fluctuating nature and peaks in the processing time in Figure 4 as some edge-addition updates require more time than others whenever an edge between two disjoint cluster of nodes comes close the propagation list grows and hence the time taken increases. Interestingly enough, for large datasets, `DBLP` and `Higgs`, the time taken to process a new edge becomes almost constant after the snapshot graphs stabilize.

The average processing time depends greatly on the characteristics of the dataset. For example, we can process `DBLP` quickly despite its size. We suspect that this is due to high diameter and high clustering coefficient.

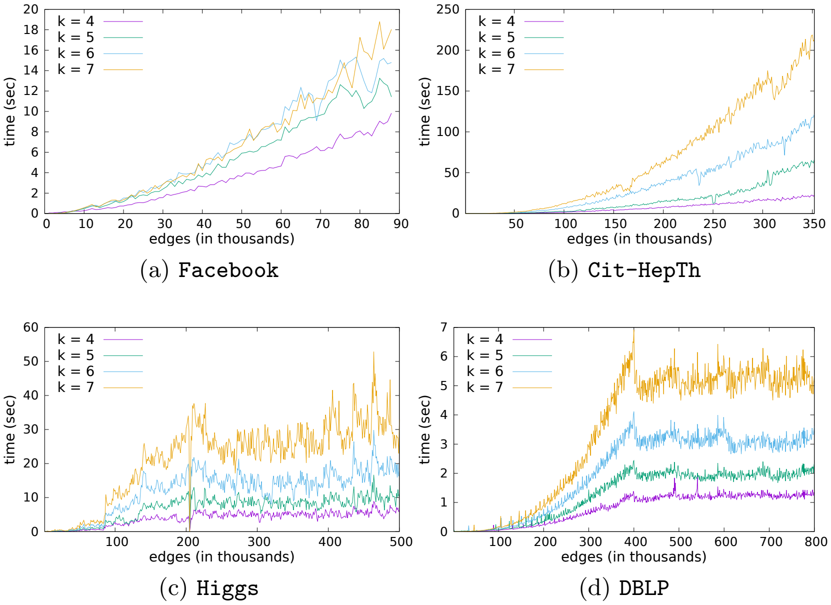
We parallelize the algorithm to measure the speed-up. In Figure 5 we see that by using 4 threads we are able to process the edges 4 times faster.

We also study the processing time as a function of the maximum distance r. Here we use `Facebook` and `DBLP`, and vary $r = 2, \ldots, 5$. The results are given in Figure 6. We see that the processing time increases exponentially as a function of r. This is expected as the neighborhood sizes also increase at a similar rate.

Space complexity: We also evaluate the memory usage of our method. The results are shown in Figure 7. Initially, the need for space increases rapidly as new nodes are added with every edge. Once all the nodes are seen the memory increase drops as only the sketches of the nodes are increasing. Note that we are not pruning any edges. As expected, the memory requirement increases linearly with ℓ.

Comparison with off-line method: Finally, for reference, we compare with a non-streaming algorithm that uses the same hyperloglog technology, the HyperANF algorithm of Boldi et al. [6]. To support querying of any window length as supported by our algorithm we modified the HyperANF algorithm to a Sliding-HyperANF algorithm by replacing the HyperLogLog sketch with Sliding HyperLogLog sketch. Running the Sliding-HyperANF algorithm in `DBLP` takes 3.6 seconds per sliding window. In contrast, for the same data-set, our streaming algorithm gives a rate of 0.003 seconds per sliding window.

[1] We measure the time for batches to get a more accurate reading.



(a) `Facebook`

(b) `Cit-HepTh`

(c) `Higgs`

(d) `DBLP`

Fig. 4. Time needed to process 1 000 edges for different ℓ

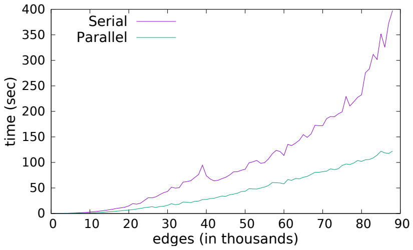
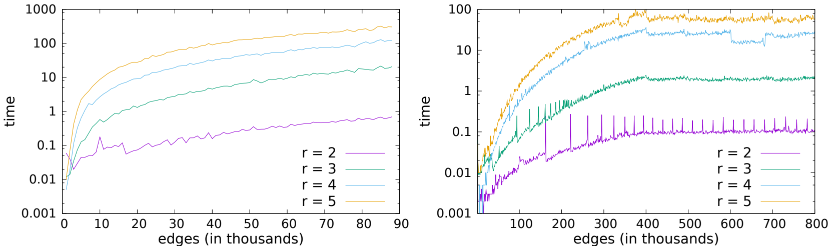


Fig. 5. Running times for `DBLP` with parallelized version of the algorithm.

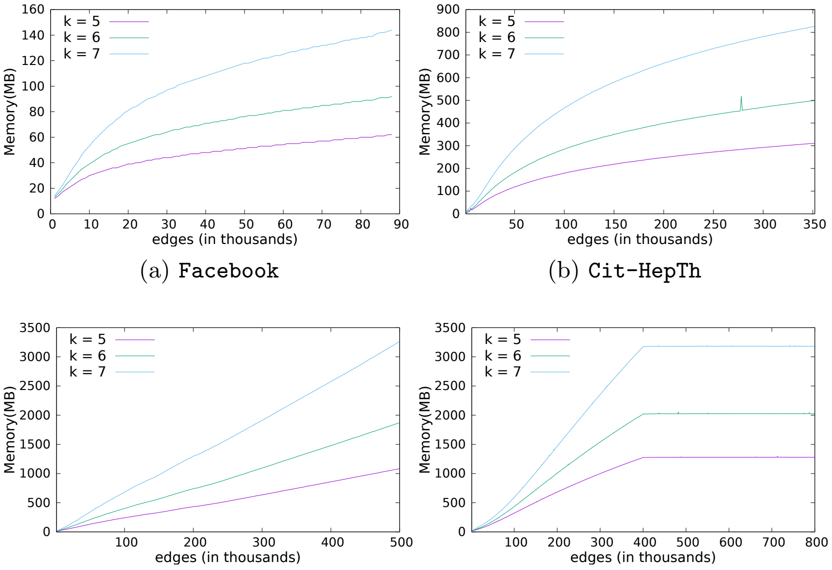
8 Concluding remarks

We studied the problem of maintaining the neighborhood profile of the nodes of an interaction network—a graph with a sequence of interactions, in the form of a stream of time-stamped edges. The model is appropriate for many modern graph datasets, like social networks where interaction between users is one of the most important aspects. We focused on the sliding-window data-stream model, which allows to forget past interactions and adapt to new drifts in the data. Thus, the proposed problem and approach can be applied to monitoring large networks with fast-evolving interactions, and used to reason how the network structure and the centrality of the important nodes change over time.



(a) `Facebook` (b) `DBLP`

Fig. 6. Time needed to process 1 000 edges as a function of distance r



(a) `Facebook` (b) `Cit-HepTh`

(c) `Higgs` (d) `DBLP`

Fig. 7. Memory utilization as a function of ℓ

We presented an exact algorithm, which is memory inefficient, but it set the stage for our main technique, an approximate algorithm based on sliding-window hyperloglog sketches, which requires logarithmic memory per network node, and has fast update time, in practice. The algorithm is also naturally parallelizable, which is exploited in our experimental evaluation to further improve its performance. One desirable property of our algorithm is that the sketch we maintain does not depend on the length of the sliding window, but the length can be specified at query time.

References

1. Ahn, K., Guha, S.: Graph sparsification in the semi-streaming model. In: Automata, Languages and Programming. pp. 328–338 (2009)
2. Arasu, A., Manku, G.: Approximate counts and quantiles over sliding windows. In: PODS. pp. 286–296 (2004)
3. Babcock, B., Datar, M., Motwani, R.: Sampling from a moving window over streaming data. In: SODA. pp. 633–634 (2002)
4. Bar-Yossef, Z., Kumar, R., Sivakumar, D.: Reductions in streaming algorithms, with an application to counting triangles in graphs. In: SODA. pp. 623–632 (2002)
5. Becchetti, L., Boldi, P., Castillo, C., Gionis, A.: Efficient semi-streaming algorithms for local triangle counting in massive graphs. In: KDD (2008)
6. Boldi, P., Rosa, M., Vigna, S.: Hyperanf: Approximating the neighbourhood function of very large graphs on a budget. In: WWW. pp. 625–634 (2011)
7. Bordino, I., Donato, D., Gionis, A., Leonardi, S.: Mining large networks with subgraph counting. In: ICDM. pp. 737–742 (2008)
8. Buriol, L., Frahling, G., Leonardi, S., Marchetti-Spaccamela, A., Sohler, C.: Counting triangles in data streams. In: PODS. pp. 253–262 (2006)
9. Chabchoub, Y., Hébrail, G.: Sliding hyperloglog: Estimating cardinality in a data stream over a sliding window. In: ICDM Workshops (2010)
10. Cohen, E.: Size-estimation framework with applications to transitive closure and reachability. Journal of Computer and System Sciences 55(3), 441–453 (1997)
11. Cohen, E.: All-distances sketches, revisited: HIP estimators for massive graphs analysis. In: PODS. pp. 88–99 (2014)
12. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms 55(1), 58–75 (2005)
13. Crouch, M., McGregor, A., Stubbs, D.: Dynamic graphs in the sliding-window model. In: ESA. pp. 337–348 (2013)
14. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining stream statistics over sliding windows. SIAM Journal on Computing 31(6), 1794–1813 (2002)
15. Durand, M., Flajolet, P.: Loglog counting of large cardinalities. In: ESA (2003)
16. Eppstein, D., Galil, Z., Italiano, G.: Dynamic graph algorithms. CRC Press (1998)
17. Flajolet, P., Fusy, É., Gandouet, O., Meunier, F.: Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. DMTCS Proceedings (2008)
18. Gama, J.: Knowledge discovery from data streams. CRC Press (2010)
19. Henzinger, M., King, V.: Randomized fully dynamic graph algorithms with polylogarithmic time per operation. Journal of the ACM 46(4), 502–516 (1999)
20. Henzinger, M., Raghavan, P., Rajagopalan, S.: Computing on data streams. In: DIMACS Workshop External Memory and Visualization. vol. 50 (1999)
21. Holme, P., Saramäki, J.: Temporal networks. Physics reports 519(3), 97–125 (2012)
22. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (Jun 2014)
23. Michail, O.: An introduction to temporal graphs: An algorithmic perspective. arXiv:1503.00278 (2015)
24. Muthukrishnan, S.: Data streams: Algorithms and applications (2005)
25. Palmer, C., Gibbons, P., Faloutsos, C.: ANF: A fast and scalable tool for data mining in massive graphs. In: KDD. pp. 81–90 (2002)
26. Rozenshtein, P., Tatti, N., Gionis, A.: Discovering dynamic communities in interaction networks. In: ECML PKDD. pp. 678–693 (2014)
27. Tsourakakis, C., Kang, U., Miller, G., Faloutsos, C.: Doulion: counting triangles in massive graphs with a coin. In: KDD. pp. 837–846 (2009)

A Correctness and complexity of the exact algorithm

Proposition 4. AddEdge *updates the summary correctly.*

Proof. Assume that we are adding $\{a, b\}$ at time t, and let H be the shapshot graph before adding this edge. Fix x. Let us define $\alpha_v(i) = h_H(x, v, i)$. Similarly, define $\beta_v(i) = h_{(t+1)}(x, v, i)$. To prove the proposition we need to show that *(1)* $\beta_v(i) = \max(g(v, x, i), \alpha_v(i))$ and *(2)* if $g(v, x, i)$ is not set, then $\alpha_v(i) = \beta_v(i)$.

Let us first prove that whenever set, we maintain the invariant,

$$g(v, x, i) \leq \max \{h(p) \mid p \in Q_v, |p| - 1 \leq i\} \leq b_v(i), \tag{1}$$

where Q_v contains all paths from x to v in $G(t + 1)$ containing (a, b) or (b, a). Note that the second inequality follows immediately from the definition of β. We prove the first by induction over i. The case $i = 1$ is trivial. If $i > 1$, then if $g(v, x, i)$ is set, then either it is set by AddEdge or there is w such that $g(w, x, i - 1)$ is set. In the first case and, due to induction assumption, in the second case, it follows that $g(v, x, i)$ is a horizon of some path in Q_v of length at most i.

We prove the main claim also by induction over i. Assume $i = 1$. The initialization of $g(v, x, 1)$ in AddEdge now guarantees *(1)* and *(2)*.

Assume $i > 1$. Assume that $\beta_v(i) > \alpha_v(i)$. This can only happen if there is a path $p = \langle v_0, \ldots, v_k\rangle \in Q_v$ with $h(p) = \beta(i)$. Let $p' = \langle v_0, \ldots, v_{k-1}\rangle$ and let $w = v_{k-1}$. We must have $\alpha_w(i - 1) < \beta_w(i - 1)$, as otherwise we have $\beta_v(i) = \alpha_v(i)$. By induction, *(1)* immediately implies that $\beta_w(i-1) = g(w, x, i-1) > \alpha_w(i-1)$. This means that Merge$(x, w, g(w, x, i - 1), i - 1)$ is called, and it returns true. Consequently, $g(v, x, i) \geq h(p) = \beta_v(i)$, Eq. 1 implies that $g(v, x, i) = \beta_v(i)$. This immediately proves *(1)* and *(2)*. □

Proposition 5. *Let* $n = |V|$*,* $m = |E|$*, and* r *be the upper bound on the distances we are maintaining. The time complexity of* AddEdge *is* $\mathcal{O}(rmn \log(n))$*. The space complexity is* $\mathcal{O}(rn^2)$*.*

Proof. The complexity of Algorithm 3 is $\log(n)$, since we need to search a summary and update $S^v[i]$ for node x.

Every $g(u, x, i + 1)$ will be initiated only if $g(v, x, i)$ was set for one of its neighbors v. As such, this may happen at most as many times as u has neighbors in the graph. Since the cummulative sum of all neighbors is $2m$ we can hence bound the number of times a $g(u, x, i + 1)$ is set for x to $2m$. Since there are n nodes, lines 5,6,7 are executed at most $2nm$ times per length i, and as a consequence this is also an upper bound on the number of calls to Algorithm 3. Putting it all together, we get a complexity of $\mathcal{O}(2nmr(\log(n)))$ for Algorithm 2. Since Algorithm 1 does only call Algorithm 2 once, this proves the complexity bound for time.

The complexity bound on space easily follows from the observation that for every node v, and every distance $i = 0, \ldots, r$, the summary $S^v[i]$ contains at most one entry for any other node. □

B Correctness and complexity of the sketch algorithm

Proposition 7 (Part 1). *The sketch version of* AddEdge *updates the summary correctly.*

Proof. Assume that we are adding $\{a, b\}$ at time t, and let H be the shapshot graph before adding this edge. Fix x. Let us define $\alpha_v(i) = t$, where $(x, t) \in C^v[i]$ (based on H), and ∞ otherwise. Similarly, define $\beta_v(i)$ using $G(t + 1)$. To prove the proposition we need to show that *(1)* $\beta_v(i) = \max(g(v, x, i), \alpha_v(i))$ and *(2)* if $g(v, x, i)$ is not set, then $\alpha_v(i) = \beta_v(i)$.

Let us first prove that whenever set, we maintain the invariant

$$g(v, x, i) \leq \max\{h(p) \mid p \in Q_v, |p| - 1 \leq i\} \leq b_v(i), \tag{2}$$

where Q_v contains all paths from a node y, with $\rho(y) = x$, to v in $G(t + 1)$ containing (a, b) or (b, a). The second inequality follows immediately by definition of β. We prove the first by induction over i. The case $i = 1$ is trivial. If $i > 1$, then if $g(v, x, i)$ is set, then either it is set by AddEdge or there is w such that $g(w, x, i - 1)$ is set. In the first case and, due to induction assumption, in the second case, it follows that $g(v, x, i)$ is a horizon of some path in Q_v of length at most i.

We prove the main claim also by induction over i. Assume $i = 1$. The initialization of $g(v, x, 1)$ in AddEdge now guarantees *(1)* and *(2)*.

Assume $i > 1$. Assume that $\beta_v(i) > \alpha_v(i)$. This can only happen if there is a path $p = \langle v_0, \ldots, v_k \rangle \in Q_v$ with $h(p) = \beta(i)$. Let $p' = \langle v_0, \ldots, v_{k-1} \rangle$ and let $w = v_{k-1}$. We must have $\alpha_w(i - 1) < \beta_w(i - 1)$, as otherwise we have $\beta_v(i) = \alpha_v(i)$. By induction, *(1)* immediately implies that $\beta_w(i-1) = g(w, x, i-1) > \alpha_w(i-1)$. This means that Merge$(x, w, g(w, x, i - 1), i - 1)$ is called, and it returns true. Consequently, $g(v, x, i) \geq h(p) = \beta_v(i)$, Eq. 2 implies that $g(v, x, i) = \beta_v(i)$. This immediately proves *(1)* and *(2)*. □

The next proposition gives an upper bound on space and memory consumption of the algorithm.

Proposition 7 (Part 2). *Let $n = |V|$, $m = |E|$, and r be the upper bound on the distances we are maintaining. The time complexity of the sketch version of* AddEdge *is $\mathcal{O}(2^k rm \log^2(n))$. The space complexity is $\mathcal{O}(2^k nr \log^2 n)$.*

Proof. Algorithm 4 needs to visit the iterate the entries in $C^v[i]$. Since there are at most $\mathcal{O}(\log n)$ different values of ρ, there are at most $\mathcal{O}(\log n)$ entries.

Every $g(u, x, i + 1)$ will be initiated only if $g(v, x, i)$ was set for one of its neighbors v. As such, this may happen at most as many times as u has neighbors in the graph. Since the cummulative sum of all neighbors is $2m$ we can hence bound the number of times a $g(u, x, i + 1)$ is set for x to $2m$. Since there are $\mathcal{O}(\log n)$ different values of ρ, lines 5,6,7 are executed at most $\mathcal{O}(\log nm)$ times per length i, and as a consequence this is also an upper bound on the number of calls to Algorithm 3. Putting it all together, we get a complexity of

$\mathcal{O}(2mr\log^2(n))$ for Algorithm 2. Since Algorithm 1 does only call Algorithm 2 once, this proves the complexity bound for time.

The complexity bound on space easily follows from the observation that for every node v, and every distance $i = 0, \ldots, r$, the summary $C^v[i]$ contains at most $\mathcal{O}(\log n)$ entries that, and each entry requires $\mathcal{O}(\log n)$ space. □

C Code to run the experiments

You can download the instruction and code from the below link:

https://github.com/rohit13k/NeighborhoodProfile.git