Continuous Similarity Computation over Streaming Graphs

Elena Valari and Apostolos N. Papadopoulos

Data Engineering Lab., Department of Informatics, Aristotle University
54124 Thessaloniki, Greece
{evalari,papadopo}@csd.auth.gr

Abstract. Large network analysis is a very important topic in data mining. A significant body of work in the area studies the problem of node similarity. One way to express node similarity is to associate with each node the set of 1-hop neighbors and compute the Jaccard similarity between these sets. This information can be used subsequently for more complex operations like link prediction, clustering or dense subgraph discovery. In this work, we study algorithms to monitor the result of a similarity join between nodes continuously, assuming a sliding window accommodating graph edges. Since the arrival of a new edge or the expiration of an existing one may change the similarity between several node pairs, the challenge is to maintain the similarity join result as efficiently as possible. Our theoretical study is validated by a thorough experimental evaluation, based on real-world as well as synthetically generated graphs, demonstrating the superiority of the proposed technique in comparison to baseline approaches.

Keywords: mining streaming graphs, continuous similarity processing.

1 Introduction

Graphs play an important role in modern world [1], due to their widespread use for modeling, representing and organizing linked data. Taking into consideration that most of the "killer" applications require a graph-based representation (e.g., the Web, social network management, protein interaction networks), efficient query processing and analysis techniques are required, not only because these graphs are massive but also because the operations that must be supported are complex, requiring significant computational resources.

A graph $G(V, E)$, in its simplest form, is composed of a node-set V, representing the entities (objects), and an edge-set E, representing the relationship among the entities. Each edge $e_{u,v} \in E$ connects a pair of nodes u, v, denoting that these nodes are directly related in a meaningful manner. For example, if nodes represent authors, then an edge between two authors may denote that they have collaborated in at least one paper. As another example, in a social network application (e.g., Facebook), an edge may denote that two users are connected by a friendship relationship.

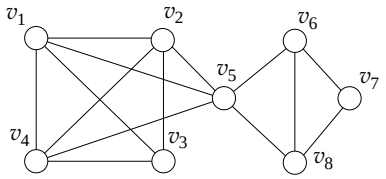


Fig. 1. Graph example.

Motivation. A significant operation in a graph is the computation of the *similarity* between nodes. The similarity between nodes u and v may be expressed in several ways, depending on application or user requirements. For example, we may express similarity by means of shortest paths, maximum flow, random walks or a combination of measures. In general, similarity is expressed by a function $V \times V \rightarrow [0, 1]$, where a value close to 0 means low similarity and a value close to 1 denotes a high similarity between a node pair. In this work, we express similarity by means of the *Jaccard similarity coefficient*, which enjoys a widespread use in diverse areas such as link prediction and recommendation [15], data cleaning [3], near duplicate detection [19], diversity analysis [9], whereas it is one of the most important measures for set similarity. We associate with each node u the set of its immediate neighbors $N(u)$ (u inclusive). Then, the similarity between nodes u and v is computed as the fraction of their common neighborhood size over the cardinality of their neighborhood union, i.e.:

$$S_J(u, v) = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|} \tag{1}$$

Example 1. Figure 1 depicts a small graph where $|V| = 8$ and $|E| = 14$. Based on our similarity definition, it can be verified easily that: $S_J(v_1, v_2) = 5/5 = 1$, $S_J(v_2, v_6) = 1/8$, $S_J(v_6, v_8) = 4/4 = 1$ and $S_J(v_1, v_7) = 0$. We observe that node pairs that share the same set of immediate neighbors (e.g., v_1 and v_2) have a similarity of 1, whereas node pairs without common neighbors have a similarity of 0 (e.g., v_1 and v_7). ■

An important operation which is based on pair-wise node similarities is the *similarity join*. More specifically, given a set of objects, a similarity function and a threshold ϑ, the similarity join operator reports all object pairs with a similarity at least ϑ. The output of this operator may be used subsequently for more complex mining tasks like clustering, dense subgraph discovery, association and link prediction. Regarding our setting, the similarity join result set R between graph nodes is defined as the set of node pairs $< u, v >$ such that the Jaccard similarity between their neighborhoods is at least ϑ. More formally:

$$R = \{(u, v) : u \in V, v \in V, S_J(u, v) \geq \vartheta\} \tag{2}$$

Our Contributions. Although similarity joins have been studied before (see for example [14]), to the best of our knowledge, there is a lack of research in maintaining the join result in a *dynamic network*, where insertions and deletions of

nodes and edges are allowed. In particular, in many modern applications, sequential access to the data is the only feasible direction, due to huge data volumes or because of frequent updates. For example, the output of a network router, in its simplest form, is usually a stream of triplets of the form $< IP_1, IP_2, t >$, denoting that IP_1 sent a packet to IP_2 at time t. Any online processing performed on the router output must be based on sequential access, since the order of the output is completely random, whereas the frequency of the stream prevents the use of expensive data structures to organize the data on-the-fly. Based on this, we assume that the graph is available in the form of a *data stream* [17], where edges should be processed as they are presented to the algorithm.

More specifically, we study two different alternatives of the *streaming graph* model. In the *turnstile model*, the graph is presented as a sequence of edge insertions and edge deletions. For example, the sequence $+e_{u,v}, +e_{u,x}, +e_{x,y}, -e_{u,x}$ represents a streaming graph which is constructed by inserting edges $e_{u,v}$, $e_{u,x}$ and $e_{x,y}$ (we use a plus sign in front of an insertion) and deleting the edge $e_{u,x}$ (we use the minus sign in front of the edge). A special case of the turnstile model is the *sliding window model*, where the last w elements are maintained in a first-in first-out fashion. In this setting, the arrival of a new edge $e_{u,v}$ is followed by the expiration of an existing edge $e_{u',v'}$. In fact, the expired edge is the one with the oldest timestamp. Based on this model, at any given time, the *active* set of edges forms a subgraph of the streaming graph, representing the last m interactions among the graph nodes. This simple model may be generalized in several directions. For example, in some cases there is a whole set of newly arrived edges, meaning that an equal number of edges must expire. Another option is to have a *time-based sliding window*, where the window maintains the interactions that took place in the last h hours. To keep the presentation and the algorithms simple, we base our work on *count-based sliding windows*, where at any given time, exactly w edges are maintained in memory, whereas arrivals and expirations refer to single edges.

The main goal of this paper is to study efficient algorithms for continuous similarity monitoring of the nodes of an evolving graph, which is presented in the form of a stream of edges. In particular, our contributions are as follows:

- To the best of our knowledge, this is the first work that studies continuous similarity computation over streaming graphs using sliding windows. Taking into consideration that node similarity is the base for more complex tasks, the results of our study can be used for clustering or community discovery over streaming graphs.
- We propose efficient algorithms to maintain the similarity join result both when insertions and deletions of edges are arbitrary and when they follow a sliding window scenario, thus, enabling the use of our techniques in any dynamic network. The proposed algorithm uses effective pruning techniques to avoid the recomputation of Jaccard similarity wherever this is possible.
- We offer a thorough experimental evaluation based on large real-world as well as synthetically generated graphs, showing that the proposed algorithms offer significant performance improvement in comparison to baseline approaches.

Roadmap. The rest of the article is organized as follows. Section 2 presents some research contributions that are highly related to our work. Algorithms for continuous similarity computation over streaming graphs are given in Section 3. Performance evaluation results based on real-world as well as synthetic networks are offered in Section 4. Finally, Section 5 concludes our work.

2 Related Work

The issues studied in this paper, lie in the intersection between graph mining [1] and data streams [11, 17]. Mining streaming graphs is challenging, mainly due to the data massiveness and also because of the inherent difficulty in solving complex graph problems in the streaming model of computation [8, 20].

Node similarity in graphs plays an important role in graph mining because it is often the base for supporting more complex operations such as clustering and community detection [10]. To express the similarity between graph nodes, a meaningful similarity measure is required. One such measure is the Jaccard similarity, which has been applied successfully in areas such as duplicate detection [6, 19], link prediction [15], similarity evaluation in wikipedia [4], triangle counting in massive graphs [5] and diversity analysis in documents [9].

Based on the importance of the Jaccard similarity, in this work we focus on the application of this measure to detect node pairs of a dynamic network, with a high degree of similarity. In particular, network dynamics are controlled by a sliding window of a fixed size w, which maintains the most recent edges of the streaming graph. Our work is inspired by previous research approaches to process complex queries over sliding window data streams. The work in [16] studies the problem of top-k query processing over a multidimensional data stream for any monotone ranking function. In a similar manner, [13] proposes efficient algorithms for top-k dominating queries whereas [12] focuses on outlier mining over general metric streams. Those works focus on multidimensional or metric streams.

Although there is a significant body of work dealing with processing over streaming graphs [8, 20, 2], none of the existing works handles similarity computation over a streaming graph using sliding windows. A research topic that is closely related to similarity computation is *triangle counting*. Algorithms for counting triangles in streaming graphs have been reported in [5] where the semi-streaming model is used, in [7] where sampling is used. Those works aim at reducing the space requirements and thus the solutions they provide are approximate. Moreover, since those techniques are based on either minhashing or sampling, they cannot support deletions efficiently.

An important challenge is that apart from the fact that, in contrast to relational join processing, the insertion/deletion of an edge affects the similarity of other node pairs in the graph, the result set is composed of two types of node pairs: i) node pairs joined by an edge and ii) non-adjacent node pairs. This feature is unique in graphs and requires attention because edge pruning cannot be performed easily.

3 Continuous Similarity Computation

3.1 Preliminaries

In this section, we present some fundamental concepts related to continuous Jaccard similarity computation in a streaming environment. Formally, the problem we attack is the following:

Problem Definition. *Given a streaming graph G and a count-based sliding window of size w, monitor all node pairs v_i, v_j such that $S_J(v_i, v_j) \geq \vartheta$, where $\vartheta \in [0, 1]$ is a user-defined similarity threshold.*

To facilitate efficient processing, the graph is organized by an adjacency list representation, where each node points to its immediate neighbors. Since in a streaming environment insertions and deletions of edges are very frequent, node information is stored in a hashmap for fast lookups. This allows us to locate each node in $O(1)$ expected time. Likewise, the result set R which contains the node pairs having similarity larger than the threshold ϑ, is also organized by a hashtable. This way, checking if a node pair is in the result set involves a lookup in the hashtable using as key a combination of the node identifiers. The indexing schemes used by our techniques are shown in Figure 2. Notice that, R may contain node pairs that either are not joined by an edge or are direct neighbors. This means that some node pairs in R correspond to adjacency list entries and some do not. For example, the entries of R shown shaded in Figure 2 correspond to disjoint node pairs, whereas the rest correspond to node pairs connected by an edge of G.

An important issue in the data organization is the way the set of neighbors $N(v)$ of a node v is arranged, since this has a direct impact on the efficiency of the Jaccard similarity computation. To provide the best possible solution we have

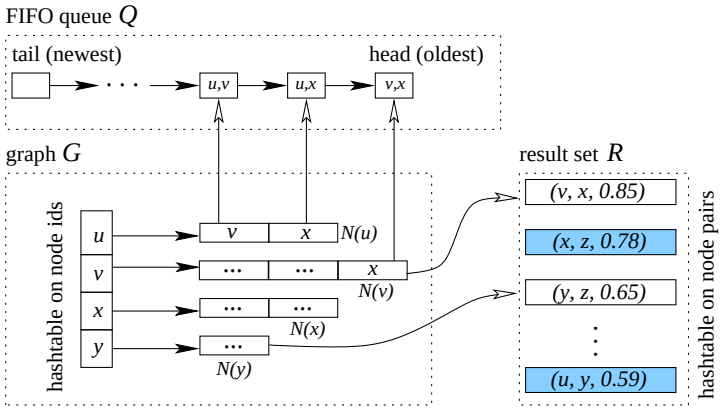


Fig. 2. Indexing techniques employed.

to take into account that: *i*) insertions and deletions in $N(v)$ must be handled efficiently and *ii*) the Jaccard similarity computation between two nodes must be also computed efficiently. We distinguish among the following cases, assuming that currently, $|N(v)| = k$:

Unordered List (UL). The set of neighbors $N(v)$ is organized as a simple unordered list. This offers $O(1)$ worst case time for inserting a new neighbor in $N(v)$, but requires linear cost to find or delete a neighbor. For Jaccard computation, if both nodes have k neighbors, then in $O(k)$ expected time we can compute the intersection and the union of the neighborhoods using hashing.

Ordered List (OL). If both lists are ordered, then the computation of the intersection and the union can be completed in $O(k)$ worst case. Likewise, insertions and deletions also require linear cost in the worst case.

Binary Search Tree (BST). With a BST, insertions and deletions are handled in $O(\log n)$ worst case, whereas intersections and union operations are executed in linear $O(k)$ time worst case.

Hash Table (HT). In this scheme, instead of having an unordered list, the set of neighbors is organized in a hashtable. This provides $O(1)$ expected cost for insertions and deletions, and also $O(k)$ expected cost for computing the intersection and the union.

Based on the previous discussion, HT is the most promising technique for Jaccard similarity computation, and this is also validated by the experimental results we report in Section 4.

3.2 Algorithmics

In this section, we study algorithmic techniques toward continuous Jaccard similarity computation over a streaming graph. Initially, we provide a baseline approach to solve the problem, followed by an efficient algorithm that can handle insertions and deletions of edges. Finally, we propose a more sophisticated algorithm which is more appropriate for the sliding-window case.

Definition 1. *The set of affected pairs of an edge $e_{u,v}$, denoted as $SAP(e_{u,v})$ or simply $SAP(u, v)$, is the set of node pairs whose Jaccard similarity is affected by the arrival or the expiration of the edge $e_{u,v}$.*

Based on the previous definition, the similarity of a node pair contained in $SAP(e_{u,v})$ may be increased or decreased, according to the structure of the graph. The following lemma explains which pairs are contained in $SAP(e_{u,v})$ and how their similarity is affected.

Lemma 1. *Let $e_{u,v}$ be an inserted/deleted edge and u, v the associated nodes. The similarities that are affected by this insertion/deletion are those defined by: i) the pair (u,v), ii) u and all neighbors of u, iii) u and all neighbors of v, iv) v and all neighbors of v and v) v and all neighbors of u.*

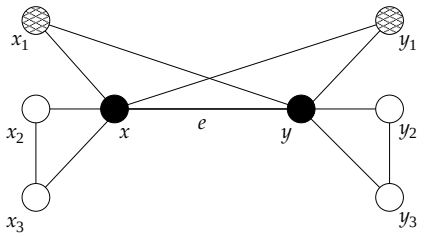


Fig. 3. Example graph used in the proof of Lemma 1.

Proof. We focus on the case where the edge e is inserted, because the deletion is handled symmetrically. Therefore, let e be a new edge joining the nodes x and y as it is indicated in Figure 3. Based on the definition of the Jaccard similarity, $SAP(x, y)$ contains only the pairs mentioned above, since by using contradiction, it is impossible that the similarity of a node pair not belonging to one of these cases will change due to the insertion of e. Next we show how the similarities of the node pairs contained in $SAP(x, y)$ are modified.

The similarity between x and y is increased, since now y becomes a direct neighbor of x and x becomes a direct neighbor of y. Consequently, the set $N(x) \cap N(y)$ gets two new members, x and y, resulting in an increase of the value of $S_J(x, y)$. Next, we examine what is the impact of inserting e to the similarity between x and each of its direct neighbors, denoted as x_i. There are two cases to examine here: in the first case, y is not a neighbor of x_i (this is the case for x_2 and x_3), whereas in the second case, y is a neighbor of x_i (e.g., when x_i is x_1). In the first case, the value of $S_J(x, x_i)$ decreases, because only the denominator increases, whereas the nominator remains unchanged. In the second case, $S_J(x, x_i)$ increases, because the nominator increases by one and the denominator remains the same. Similar arguments can be stated for the other node pairs contained in $SAP(x, y)$. □

The Baseline Algorithm (BASE). The simplest algorithm to solve the continuous similarity problem is directly derived by utilizing the result of Lemma 1. This baseline algorithm, denoted as BASE, computes the Jaccard similarity for all node pairs contained in $SAP(x, y)$, where x and y are the nodes associated to an edge e which is either inserted or deleted. Each time a new similarity $S_J(u, v)$ is computed, the value is compared to ϑ, and if $S_J(u, v) \geq \vartheta$, then the pair (u, v) is inserted into the result set R. Node pairs that are contained in R and their updated similarity is less than ϑ are simply evicted from R.

It is evident, that the cost of this approach is highly dependent on the number of Jaccard similarity computations executed. To reduce this number, we first enforce an upper bound, and if the node pair still survives the test, only then the Jaccard similarity is computed. In particular, given two nodes u and v, their

Algorithm 1 BASE

Input: G: the graph, e: the new or expiring edge between x and y, R: result set
Output: updated result set R

```
1: determine the set SAP(x, y);
2: for each node pair (u, v) ∈ SAP(x, y)
3:     compute UB_J(u, v)
4:     if (UB_J(u, v) < ϑ)
5:         if ((u, v) ∈ R) R ← R − {(u, v)};
6:     else
7:         recompute S_J(u, v);
8:         if (S_J(u, v) ≥ ϑ)
9:             if ((u, v) ∉ R) R ← R + {(u, v)}; /* insert (u, v) into R */
10:        else
11:            if ((u, v) ∈ R) R ← R − {(u, v)}; /* remove (u, v) from R */
12: return;
```

Jaccard similarity satisfies the following inequality:

$$S_J(u,v) \leq UB_J(u,v) = \frac{min(|N(u)|, |N(v)|)}{max(|N(u)|, |N(v)|)} \tag{3}$$

The outline of BASE is given in Algorithm 1. The upper bound test is performed at Line 4, and the Jaccard computation is executed at Line 7. Although the use of the upper bound reduces the number of Jaccard similarity computations, more sophisticated techniques are required to decrease the cost further.

The Counter-based Algorithm (COUNTER). The main drawback of BASE is that there is a significant number of Jaccard similarity computations, leading to performance deterioration. To overcome this limitation, the next algorithm (COUNTER) is based on keeping separate counters for the cardinality of the intersection (nominator) and the cardinality of the union (denominator), thus reducing the cost of computing Jaccard similarities significantly.

The key idea of the COUNTER algorithm is that when a new edge e joining x and y is inserted, we compute the value $S_J(x,y)$ and we maintain two separate counters $C_\cap$ and $C_\cup$ for the cardinality of the intersection and the union of the neighborhoods respectively, i.e. $C_\cap(x,y) = |N(x) \cap N(y)|$ and $C_\cup(x,y) = |N(x) \cup N(y)|$. Thus, whenever there is a need to recompute the value of $S_J(x,y)$, we need only adjust the values of $C_\cap(x,y)$ and $C_\cup(x,y)$ and just perform the division $C_\cap(x,y)/C_\cup(x,y)$. In addition, intersection and union counters are maintained for node pairs that are not connected by an edge but are included in the result set R. Subsequent recomputations of the Jaccard similarity are executed fast, avoiding unnecessary set-oriented operations among the neighborhoods. In the sequel, we examine only the insertion case, since deletions are handled in a similar manner.

Algorithm 2 COUNTER

Input: G: the graph, e: the new edge between x and y, R: result set
Output: updated result set R

```
1: determine the set SAP(x, y);
2: for each node pair (u, v) ∈ SAP(x, y)
3:     if ((u, v) ∈ R)
4:         update counters C∩(u, v) and C∪(u, v) for (u, v);
5:         if (C∩(u, v)/C∪(u, v) < ϑ)
6:             R ← R − {(u, v)};
7:     else if ((u, v) ∈ E) /* edge (u, v) exists */
8:         update counters C∩(u, v) and C∪(u, v) for (u, v);
9:         if (C∩(u, v)/C∪(u, v) ≥ ϑ)
10:            R ← R + {(u, v)};
11:    else
12:        compute UB_J(u, v)
13:        if (UB_J(u, v) ≥ ϑ)
14:            compute S_J(u, v);
15:            if (S_J(u, v) ≥ ϑ)   R ← R + {(u, v)};
16: return;
```

Let e be a new edge that is inserted in G, linking nodes x and y. COUNTER first checks if the pair (x, y) is already in the result set R. If yes, then definitely there exist counters for the intersection and the union that have been set previously. Therefore, the new value of $S_J(x, y)$ is computed easily. The outline of COUNTER is given in Algorithm 2. Notice that, before the computation of the Jaccard similarity in Line 12, the algorithm first checks if the node pair is in R or the corresponding edge exists in E. Then, the intersection and union counters are updated based on the cases reported in Lemma 1. To avoid confusion, we use the term *set-based Jaccard computation* to refer to the Jaccard computation when there are no precomputed counters, and use the term *counter-based Jaccard computation* otherwise.

The Slide-oriented Algorithm (SLIDE). Although COUNTER is more efficient than BASE, it is designed to support insertion and deletion of arbitrary edges. However, our goal is to support continuous evaluation over a sliding window of size w. In this case, we know exactly the expiration time of an edge, since edges arrive and depart in a FIFO fashion. This means that additional optimizations can be applied toward the design of an algorithm which is more appropriate for the sliding-window case.

In this section, we provide the details of the SLIDE algorithm, which has been designed for the sliding-window scenario. The key idea of SLIDE is that if we could determine the time instance when a node pair (u, v) will enter the result set R, then we could decide if (u, v) is promising or not. It turns out that such a prediction is possible, resulting in an effective mechanism to determine

node pairs that can be eliminated safely. More specifically, when a new edge is inserted, we make an optimistic prediction, determining the closer time instance that the nodes associated with the edge can be included in the result set R. The prediction is optimistic, in the sense that the estimated time instance is computed assuming the best possible scenario for this edge. In addition, as we show in the sequel, this estimation produces only false positives and never false dismissals.

Lemma 2. *Let e be a newly arrived edge joining nodes x and y. Let $t^*(x, y)$ denote the closer time instance into the future in order for (x, y) to enter the result set R. Then, it holds that:*

$$t^*(x, y) = t_{now} + \min\{C^*_\cap(e) - C_\cap(e), C_\cup(e) - C^*_\cup(e)\} \tag{4}$$

where t_{now} is the current time, $C_\cap(e)$ and $C_\cup(e)$ are the values of intersection and union counters computed upon examination of e, and $C^*_\cap(e)$ and $C^*_\cup(e)$ are the values of intersection and union counters when e is expected to be inserted into R.

Proof. Assume that e joins the nodes x and y and it is checked at the current time t_{now}. Let also $C_\cap(e)$ and $C_\cup(e)$ denote the values of the intersection and union counters for e at time t_{now}. We assume that the node pair (x, y) will enter the result set R at some time in the future, and let $t^*(x, y)$ denote this particular time instance. We are looking for the smallest possible value of $t^*(x, y)$. If $C^*_\cap(e)$ and $C^*_\cup(e)$ are the values of the intersection and union counters when (x, y) enters R, then clearly we have that: $C^*_\cap(e)/C^*_\cup(e) \geq \vartheta$. Based on the previous discussion, every time $S_J(x, y)$ is affected, exactly one of the following five cases is true: i) only $C_\cap(e)$ increases, ii) only $C_\cup(e)$ increases, iii) only $C_\cap(e)$ decreases, iv) only $C_\cup(e)$ decreases, iv) both $C_\cap(e)$ and $C_\cup(e)$ increase or v) both $C_\cap(e)$ and $C_\cup(e)$ decrease. Among the previous cases, the ones that may lead faster to the inclusion of (x, y) into R are the first two. Indeed, the fraction $C_\cap(e)/C_\cup(e)$ increases faster if either the nominator increases (keeping the denominator fixed) or the denominator decreases (keeping the nominator fixed). Note, that these two events cannot happen at the same time. Consequently, to gain the additional $\Delta\vartheta$ similarity score required to enter R, it suffixes to wait for $min(C^*_\cap(e) - C_\cap(e), C_\cup(e) - C^*_\cup(e))$ time instances at best, assuming the most optimistic scenario. □

Lemma 3. *If for an edge e it holds that $t_{exp}(e) < t^*(e)$, then it is safe to skip the Jaccard similarity computation for this edge.*

Proof. Recall that $t^*(e)$ is the closest time instance when e will enter R, considering the most favorable scenario for e, i.e., by increasing the nominator and decreasing the denominator as much as possible. Consequently, if the expiration time of e is less than $t^*(e)$, then it is impossible for e to enter the result set R. Therefore, the Jaccard computation between the nodes joined by e may be skipped safely. □

Algorithm 3 SLIDE

Input: G: the graph, e: the new edge between x and y, R: result set
Output: updated result set R

```
1: determine the set SAP(x, y);
2: for each node pair (u, v) ∈ SAP(x, y)
3:     if ((u, v) ∈ R)
4:         update counters C∩(u, v) and C∪(u, v) for (u, v);
5:         if (C∩(u, v)/C∪(u, v) < ϑ)
6:             R ← R − {(u′, v′)};
7:     else if ((u, v) ∈ E) /* edge (u, v) exists */
8:         if  (t_exp(u, v) < t*(u, v))
9:             reject (u, v) from further consideration;
10:        else
11:            update counters C∩(u, v) and C∪(u, v) for (u, v);
12:            if (C∩(u, v)/C∪(u, v) ≥ ϑ)
13:                R ← R + {(u, v)};
14:    else
15:        compute UB_J(u, v)
16:        if (UB_J(u, v) ≥ ϑ)
17:            compute S_J(u, v);
18:            if (S_J(u, v) ≥ ϑ)   R ← R + {(u, v)};
19: return;
```

If an edge e joining nodes x and y satisfies the inequality of Lemma 3, then there is no need to test the pair (x, y) again, and consequently there is no need to maintain intersection and union counters, since it is guaranteed that e will never enter R for the rest of its lifespan. The outline of SLIDE is given in Algorithm 3. The expiration time pruning is applied in Lines 8 and 9.

4 Performance Evaluation

In this section, we report some representative performance results showing the efficiency and scalability of the proposed approach. All algorithms have been implemented in JAVA and the experiments have been conducted on an Intel Core i5@2.7GHz machine. We study the performance of the algorithms in terms of their runtime and their pruning capabilities, by varying the most important parameters, such as the window size (w) and the value of the similarity threshold (ϑ). The default values for the parameters, if not otherwise specified, are: $w = 1,000,000$ and $\vartheta = 0.8$. The computational cost of the algorithms is given in terms of the expected time required by an update (an insertion followed by a deletion). This value determines the processing capabilities of the algorithms, since it is inversely proportional to the number of updates that can be served per time unit, which is an important measure in applications managing data streams.

4.1 Data Description

To study the performance of the algorithms we have used both real-world and synthetic data sets. The real-world data sets are described briefly in Table 1 and are freely available for download at http://snap.stanford.edu/data/index.html.

Table 1. Real-world data sets (*source http://snap.stanford.edu/data/index.html).*

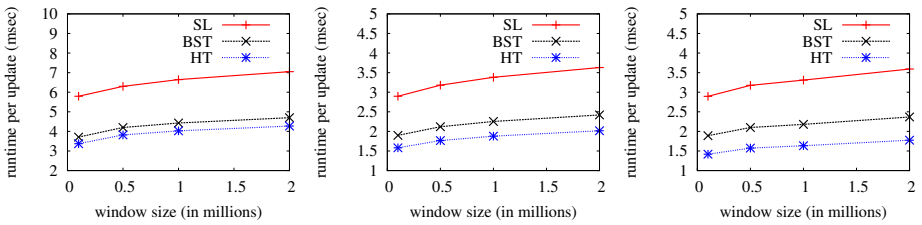
Data	#Nodes	#Edges	Description
Wiki-Talk	2,394,385	5,021,410	pages editing between wikipedia users
Web-BerkStan	685,230	7,600,595	web from `berkeley.edu` and `stanford.edu`
Soc-LiveJournal1	4,847,571	68,993,773	users' connections in LiveJournal social network

The synthetic graphs have been generated by using the GenGraph tool [18]. This generator produces graphs obeying power-law degree distributions. In particular, GenGraph generates a set of n integers in the interval $[d_{min}, d_{max}]$ obeying a power-law distribution with exponent a. Therefore, according to the degree distribution produced, a random power-law graph is generated. The default values for the parameters of the generator are: $d_{min} = 0.1\%$ of the number of vertices, $d_{max} = 0.8\%$ of the number of vertices and $a \in \{1.8, 2, 2.2, 2.5\}$. The maximum number of vertices has been set to 10,000.

4.2 Experimental Results for Real-life Data

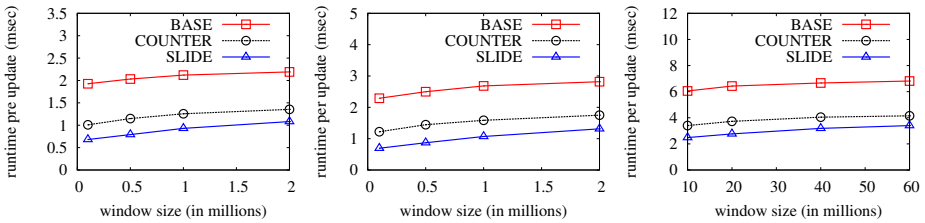
The first result involves the way Jaccard computations are computed, which is highly related to the way adjacency lists are maintained, as it has been described in Section 3.1. In particular, Figure 4 depicts the performance of the three studied algorithms for the Soc-LiveJournal1 data set. As expected, the HT organization, which relies on hashing, shows the best performance. Therefore, we apply the HT technique in the performance evaluation discussed in the sequel.

Figures 5 and 6 demonstrate the scalability of the algorithms by varying the windows size w. Figure 5 shows the runtime per update. All algorithms



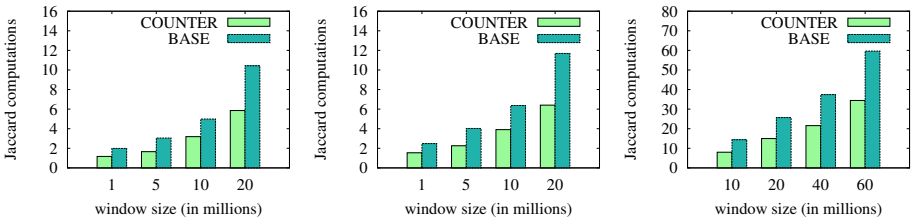
(a) Base (b) Counter (c) Slide

Fig. 4. Comparison of adjacency list maintenance using Soc-LiveJournal1.



(a) Wiki-Talk (b) Web-BerkStan (c) Soc-LiveJournal1

Fig. 5. Runtime vs window size.



(a) Wiki-Talk (b) Web-BerkStan (c) Soc-LiveJournal1

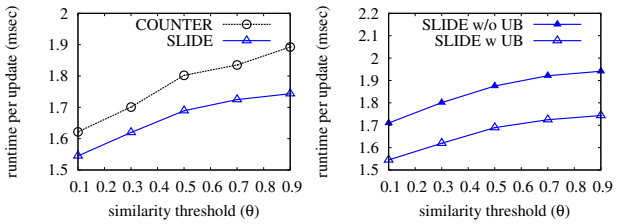
Fig. 6. Jaccard similarity computations vs window size.

Table 2. Number of counter-based Jaccard computations (Soc-LiveJournal1).

threshold ϑ	result set R	SLIDE executed	SLIDE saved	COUNTER executed
0.1	294,117	308,011,791	8,913,518	316,925,309
0.3	179,213	321,129,674	16,828,460	337,958,134
0.5	98,173	333,849,425	27,991,030	361,840,455
0.7	63,171	341,328,740	36,990,891	378,319,631
0.9	304	350,187,993	45,882,785	396,070,778

are affected negatively when the number of active edges increases. However, we observe that COUNTER and SLIDE are consistently more efficient than BASE. This is explained by studying the number of Jaccard computations performed by each algorithm. Figure 6 compares BASE and COUNTER with respect to the number of similarity computations. It is evident, that the counter-based technique employed by COUNTER saves a significant number of set-based similarity computations, which is the predominant cost in runtime. In general, SLIDE is around four times faster than BASE and two times faster than COUNTER, despite the fact that the upper bound pruning is enabled for all algorithms.

Next, we illustrate the impact of the similarity threshold to the performance of the algorithms. For this, we have used our largest graph, i.e., Soc-LiveJournal1. Table 2 shows the number of counter-based Jaccard computations executed by COUNTER and SLIDE. Although both algorithms execute the same number of set-



(a) COUNTER vs SLIDE (b) UB pruning

Fig. 7. Performance vs threshold (Soc-LiveJournal1)

based Jaccard computations, SLIDE manages to reduce the number of counter-based Jaccard computations due to the expiration time pruning technique employed. This leads to a significant performance improvement.

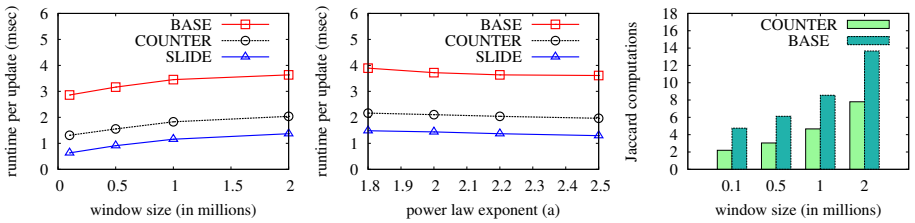
Figure 7(a) shows the runtime comparison between COUNTER and SLIDE algorithms. We observe that the performance gap between the algorithms increases by increasing the similarity threshold. Note that, as it is also shown in Table 2, the larger the similarity threshold the fewer node pairs manage to enter the result set. This means that we are going to have less precomputed information in R and therefore SLIDE benefits more by this situation since it can skip more counter-based Jaccard computations.

Finally, in Figure 7(b) we report on the pruning power of the upper bound given in Equation 3, when applied to the SLIDE algorithm. We clearly see that there is a performance gain ranging between 12% and 20%, which is very important, since the runtime per update defines the throughput (edges per time unit) that can be processed by the algorithm.

4.3 Experimental Results for Synthetic Data

In the sequel, we report some evaluation results showing the efficiency of the proposed approach over synthetic streaming graphs. These graphs in which the experiments were performed are denser than the real-life graphs explored previously. In particular, as the value of parameter a (power-law exponent) decreases, the graph generated by GenGraph [18] contains more nodes with large degree, resulting in a graph with larger density. This means that the density of a graph with $a = 1.8$ is larger than that of a graph with $a = 2.2$.

Figure 8 shows the performance of the algorithms, for different values of the window size w and the power-law exponent a. Again, as in the case of real-world data, we observe that SLIDE shows the best performance in terms of runtime (Figure 8(a)) and this is also true for different values of the power-law exponent (Figure 8(b)). The small performance difference of all algorithms when the power-law exponent increases, is due to the impact of a on the graph density, because the cardinality of the SAP of a node pair is highly dependent on the density of the graph. The number of set-based Jaccard computations are given



(a) runtime vs w (b) runtime vs a (c) Jaccard computations

Fig. 8. Performance for synthetic data sets.

in Figure 8(c). Again, the precomputed counters save a significant number of set-based Jaccard computations, resulting in performance improvement.

5 Conclusions

Node similarity in graphs is an important operation, because it allows the execution of more complex analysis tasks such as clustering and community discovery. In this work, we have studied algorithms for continuous evaluation of pair-wise similarities, where the graph is accessed as a random sequence of edges in a sliding window scenario. More specifically, given a similarity threshold ϑ, we are interested in determining all node pairs with Jaccard similarity at least ϑ. This problem arises frequently in data streams, and especially in streaming graphs, where a sliding window retains the last w entity interactions.

Three algorithms have been studied and evaluated, namely Base, which is the baseline approach, Counter an algorithm that supports insertion and deletion of any edge and it is based on precomputed counters and finally Slide which is designed for a streaming scenario and uses a pruning technique to ignore node pairs that will never make it to the result set. Experimental results based on real-world and synthetic data sets have demonstrated that Slide is consistently more efficient that the other algorithms.

There are several interesting directions for future work, such as: i) the design of algorithm for top-k most similar pairs, ii) the generalization of our techniques to consider h-hop neighbors for similarity computation and iii) the use of sketch-based techniques to enable performance boost by penalizing the accuracy of the result. With respect to the last direction, graph-specific sketches, like the gSketch [21] or cascading summaries [8], may be applied to allow for low-space similarity computation.

References

1. C. Aggarwal, H. Wang. *Managing and Mining Graph Data*, Springer, 2010.
2. C. Aggarwal, Y. Zhao, P. S. Yu. On Clustering Graph Streams. *Proceedings of SIAM SDM*, pp.478-489, 2010.

3. A. Arasu, V. Ganti, R. Kaushik. Efficient Exact Set-Similarity Joins. *Proceedings of VLDB*, pp.918-929, 2006.
4. J. Bank and B. Cole, Calculating the Jaccard Similarity Coefficient with Map Reduce for Entity Pairs in Wikipedia, http://weblab.-infosci.cornell.edu/weblab/papers/Bank2008.pdf, 2008.
5. L. Becchetti, P. Boldi, C. Castillo, A. Gionis. Efficient Semi-Streaming Algorithms for Local Triangle Counting in Massive Graphs. *Proceedings of ACM SIGKDD*, pp.16-24, 2008.
6. A. Z. Broder, S. C. Glassman, M. S. Manasse, G. Zweig. Syntactic clustering of the web. *Proceedings of WWW*, pp.1157-1166, 1997.
7. L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, C. Sohler. Counting triangles in data streams. *Proceedings of ACM PODS*, pp.253-262, 2006.
8. G. Cormode, S. Muthukrishnan. Space Efficient Mining of Multigraph Streams. *Proceedings of ACM PODS*, pp.271-282, 2005.
9. F. Deng, S. Siersdorfer, S. Zerr. Efficient Jaccard-based Diversity Analysis of Large Document Collections. *Proceedings of ACM CIKM*, pp.1402-1411, 2012.
10. S. Fortunato. Community Detection in Graphs. arXiv:0906.0612v2 [physics.soc-ph], 2009.
11. M. Garofalakis, J. Gehrke, R. Rastogi. Querying and Mining Data Streams: You Only Get One Look. *Proceedings of ACM SIGMOD*, tutorial, 2002.
12. M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, Y. Manolopoulos. Continuous Monitoring of Distance-Based Outliers over Data Streams. *Proceedings of IEEE ICDE*, pp.135-146, 2011.
13. M. Kontaki, A. N. Papadopoulos, Y. Manolopoulos. Continuous Top-k Dominating Queries. *IEEE Transactions on Knowledge and Data Engineering*, 24(5), pp.840-853, 2012.
14. X. Lian, L. Chen. Efficient Join Processing on Uncertain Data Streams. *Proceedings of ACM CIKM*, pp.857-866, 2009.
15. D. Liben-Nowell, J. Kleinberg. The Link Prediction Problem for Social Networks. *Proceedings of ACM CIKM*, pp. 556-559, 2003.
16. K. Mouratidis, S. Bakiras, D. Papadias. Continuous Monitoring of Top-k Queries over Sliding Windows. *Proceedings of ACM SIGMOD*, pp.635-646, 2006.
17. S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), pp.117-236, 2005.
18. F. Vigen, M. Latapy. Efficient and Simple Generation of Random Simple Connected Graphs with Prescribed Degree Sequence. *Proceedings of COCOON*, pp.440-449, 2005.
19. C. Xiao, W. Wang, X. Lin, J. X. Yu, G. Wang. Efficient Similarity Joins for Near Duplicate Detection. *ACM Transactions on Database Systems*, 36(3), 15:1-15:41, 2011.
20. M. Zelke. Algorithms for Streaming Graphs. *PhD Dissertation, Humboldt University of Berlin*, 2009.
21. P. Zhao, C. C. Aggarwal, M. Wang. gSketch: On Query Estimation in Graph Streams *Proceedings of VLDB*, pp.193-204, 2011.