Proximity Tracking on Time-Evolving Bipartite Graphs

Hanghang Tong* Spiros Papadimitriou† Philip S. Yu‡ Christos Faloutsos*

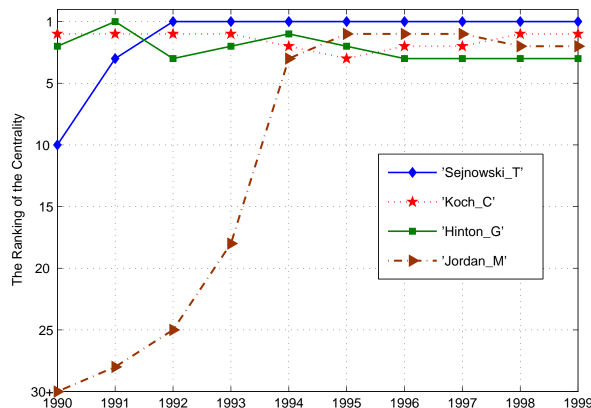
Abstract
Given an author-conference network that evolves over time, which are the conferences that a given author is most closely related with, and how do they change over time? Large time-evolving bipartite graphs appear in many settings, such as social networks, co-citations, market-basket analysis, and collaborative filtering.

Our goal is to monitor (i) the centrality of an individual node (e.g., *who are the most important authors*?); and (ii) the proximity of two nodes or sets of nodes (e.g., *who are the most important authors with respect to a particular conference*?) Moreover, we want to do this efficiently and incrementally, and to provide "any-time" answers. We propose **pTrack** and **cTrack**, which are based on random walk with restart, and use powerful matrix tools. Experiments on real data show that our methods are effective and efficient: the mining results agree with intuition; and we achieve up to *15∼176 times* speed-up, without any quality loss.

1 Introduction

Measuring proximity (a.k.a relevance) between nodes on bipartite graphs (see [18] for the formal definition of bipartite graph) is a very important aspect in graph mining and has many real applications, such as ranking, spotting anomaly nodes, connection subgraphs, pattern matching and many more (see Section 7 for a detailed review).

Despite their success, most existing methods are designed for static graphs. In many real settings, the graphs are evolving and growing over time, e.g. new links arrive or link weights change. Consider an author-conference evolving graph, which effectively contains information about the number of papers (edge weights) published by each author (type 1 node) in each conference (type 2 node) for each year (timestamp). Trend analysis tools are becoming very popular. For example, Google Trends[1] provides useful insights, despite the simplicity of its approach. For instance, in the setting of our example, a tool similar to Google Trends might answer questions such as "*How does the number of papers published by an author vary over time?*" or "*How does the number of papers published in a particular conference or research area (i.e., set of conferences) vary over time?*" This kind of analysis takes into account paper counts for either



(a) The ranking of centrality for some authors in NIPS.

1992	1997	2002	2007
ICDE	CIKM	KDD	ICDM
ICDCS	ICDCS	SIGMOD	KDD
SIGMETRICS	ICDE	ICDM	ICDE
PDIS	SIGMETRICS	CIKM	SDM
VLDB	ICMCS	ICDCS	VLDB

(b) Philip S. Yu's top 5 conferences at four time steps, using a window of 5 years.

Figure 1: Scaling sophisticated trend analysis to time-evolving graphs. See Section 6 for detailed description of results.

an author or a conference alone or, at best, a single, specific author-conference pair. Instead, we want to employ powerful analysis tools inspired by the well-established model of random walk with restart to analyze the entire graph and provide further insight, taking into account all author-conference information so far, i.e., including indirect relationships among them. However, if we need to essentially incorporate all pairwise relationships in the analysis, scalability quickly becomes a major issue. This is precisely the problem we address in this paper: how can we efficiently keep track of proximity and avoid global re-computation as new information arrives. Fig. 1 shows examples of our approach.

In this paper, we address such challenges in multiple dimensions. In particular, this paper addresses the following questions:

Q1: How to define a good proximity score in a dynamic setting?

Q2: How to incrementally track the proximity scores between nodes of interest, as edges are updated?

Q3: What data mining observations do our methods enable?

*Carnegie Mellon University
†IBM T.J. Watson Lab
‡University of Illinois at Chicago

[1] `http://www.google.com/trends/`

We begin in Section 2 with the problem definition and, in Section 3, we propose our proximity definition for dynamic bipartite graphs. We carefully design our measurements to deal with (1) the links arriving at different time steps and (2) important properties, such as monotonicity. Proximity will also serve as the basis of our centrality measurement in the dynamic setting. Then, in Section 4, we study computational issues thoroughly and propose two fast algorithms, which are the core of computing our dynamic proximity and centrality measurements. The complete algorithms to track proximity (*Track-Proximity*) and centrality (*Track-Centrality*) are presented in Section 5. In Section 6, we verify the effectiveness and efficiency of our proposed dynamic proximity on real datasets.

The major contributions of the paper can be summarized as follows:

1: Definitions of proximity and centrality for time-evolving graphs.

2: Two fast update algorithms (*Fast-Single-Update* and *Fast-Batch-Update*), without any quality loss.

3: Two algorithms to incrementally track centrality (*Track-Centrality*) and proximity (*Track-Proximity*) in any-time fashion.

4: Extensive experimental case-studies on several real datasets, showing how different queries can be answered, achieving up to **15∼176x** speed-up.

2 Problem Definitions

Table 2 lists the main symbols we use throughout the paper. Following standard notation, we use capital letters for matrices $\mathbf{M}$, and arrows for vectors. We denote the transpose with a prime (i.e., $\mathbf{M}'$ is the transpose of $\mathbf{M}$), and we use parenthesized superscripts to denote time (e.g., $\mathbf{M}^{(t)}$ is the time-aggregate adjacency matrix at time t). When we refer to a static graph or, when time is clear from the context, we omit the superscript (t). We use subscripts to denote the size of matrices/vectors (e.g. $\mathbf{0}_{n\times l}$ means a matrix of size $n \times l$, whose elements are all zero). Also, we represent the elements in a matrix using a convention similar to Matlab, e.g., $\mathbf{M}(i,j)$ is the element at the ith row and j^{th} column of the matrix $\mathbf{M}$, and $\mathbf{M}(i,:)$ is the ith row of $\mathbf{M}$, etc. Without loss of generality, we assume that the numbers of type 1 and type 2 objects are fixed (i.e., n and l are constant for all time steps); if not, we can reserve rows/columns with zero elements as necessary.

At each time step, we observe a set of new edges or edge weight updates. These represent the link information that is available at the finest time granularity. We use the *time-slice matrix*, or ***slice matrix*** for brevity, $\mathbf{S}^{(t)}$ to denote the new edges and additional weights that appear at time step t. For example, given a set of authors and annual conferences, the number of papers that author i publishes in conference j during year t is the entry $\mathbf{S}^{(t)}(i,j)$. In this paper, we focus

Table 1: Symbols

Symbol	Definition and Description
$\mathbf{M}^{(t)}$	$n \times l$ time-aggregate adjacency matrix at time t
$\mathbf{S}^{(t)}$	$n \times l$ slice matrix at time t
$\Delta\mathbf{M}^{(t)}$	$n \times l$ difference matrix at time t
$\mathbf{D}_1^{(t)}$	$n \times n$ out-degree matrix for type 1 object, i.e. $\mathbf{D}_1^{(t)}(i,i) = \sum_{j=1}^{n} \mathbf{M}^{(t)}(i,j)$, and $\mathbf{D}_1^{(t)}(i,j) = 0\ (i \neq j)$
$\mathbf{D}_2^{(t)}$	$l \times l$ out-degree matrix for type 2 object, $i.e.\mathbf{D}_2^{(t)}(i,i) = \sum_{j=1}^{n} \mathbf{M}^{(t)}(j,i)$, and $\mathbf{D}_2^{(t)}(i,j) = 0\ (i \neq j)$
$\mathbf{I}$	identity matrix
$\mathbf{0}$	a matrix with all elements equal to 0
$\mathbf{1}$	a matrix with all elements equal to 1
n, l	number of nodes for type 1 and type 2 objects, respectively ($n > l$)
m	number of edges in the bipartite graph
c	$(1-c)$ is fly-out probability for random walk with restart (set to be 0.95 in the paper)
$r_{i,j}^{(t)}$	proximity from node i to node j at time t

only on the case of edge additions and weight increases (e.g., authors always publish new papers, and users always rate more movies). However, the ideas we develop can be easily generalized to handle other types of link updates, such as links deletions or edge weights decreases.

Given the above notion, a dynamic, evolving graph can be naturally defined as a sequence of observed new edges and weights, $\mathbf{S}^{(1)}, \mathbf{S}^{(2)}, \ldots, \mathbf{S}^{(t)}, \ldots$. However, the information for a single time slice may be too sparse for meaningful analysis, and/or users typically want to analyze larger portions of the data to observe interesting patterns and trends. Thus, from a sequence of slice matrices observed so far, $\mathbf{S}^{(j)}$ for $1 \leq j \leq t$, we construct a bipartite graph by aggregating time slices. We propose three different aggregation strategies, which place different emphasis on edges based on their age. In all cases, we use the term *time-aggregate adjacency matrix* (or ***adjacency matrix*** for short), denoted by $\mathbf{M}^{(t)}$, for the adjacency matrix of the bipartite graph at time step t. We will introduce the aggregation strategies in the next section).

Finally, to simplify the description of our algorithms, we introduce the ***difference matrix*** $\Delta\mathbf{M}^{(t)}$, which is the difference between two consecutive adjacency matrices, i.e., $\Delta\mathbf{M}^{(t)} \triangleq \mathbf{M}^{(t)} - \mathbf{M}^{(t-1)}$. Note that, depending on the aggregation strategy, difference matrix $\Delta\mathbf{M}^{(t)}$ may or may not be equal to the slice matrix $\mathbf{S}^{(t)}$.

An important observation from many real applications is that, despite the large size of the graphs involved (with hundreds of thousands or millions of nodes and edges), the intrinsic dimension (or, effective rank) of their corresponding adjacency matrices is usually relatively small, primarily

because there are relatively fewer objects of one type. For example, on the author-conference graph from the **AC** dataset (see Section 6), although we have more than 400,000 authors and about 2 million edges, with only ~ 3500 conferences. In the user-movie graph from the **NetFlix** dataset, although we have about 2.7 million users with more than 100 million edges, there are only 17,700 movies. We use the term *skewed* to refer to such bipartite graphs, i.e., $n, m \gg l$.

With the above notation, our problems (**pTrack** and **cTrack**) can be formally defined as follows:

PROBLEM 1. ***pTrack***

Given: *(i) a large,skewed time-evolving bipartite graph* $\{\mathbf{S}^{(t)}, t = 1, 2, ...\}$*, and (ii) the query nodes of interest* $(i, j, ...)$

Track: *(i) the top-k most related objects for each query node at each time step; and (ii) the proximity score (or the proximity rank) for any two query nodes at each time step.*

There are two different kinds of tracking tasks in **pTrack**, both of which are related to proximity. For example, in a time-evolving author-conference graph we can track "*What are the major conferences for John Smith in the past 5 years?*" which is an example of task (i); or "*How much credit (importance) has John Smith accumulated in the KDD Conference so far?*" which is an example of task (ii). We will propose an algorithm (*Track-Proximity*) in Section 5 to deal with **pTrack**.

PROBLEM 2. ***cTrack***

Given: *(i) a large,skewed time-evolving bipartite graph* $\{\mathbf{S}^{(t)}, t = 1, 2, ...\}$*, and (ii) the query nodes of interest* $(i, j, ...)$

Track: *(i) the top-k most central objects in the graph, for each query node and at each time step; and (ii) the centrality (or the rank of centrality), for each query node at each time step.*

In **cTrack**, there are also two different kinds of tracking tasks, both of which are related to centrality. For example, in the same time-evolving author-conference graph, we can track "*How influential is author-A over the years?*" which corresponds to task (i); or "*Who are the top-10 influential authors over the years?*" which corresponds to task (ii). Note that in task (i) of **cTrack**, we do not need the query nodes as inputs. We will propose another algorithm (*Track-Centrality*) in Section 5 to deal with **cTrack**.

For all these tasks (**pTrack** and **cTrack**), we want to provide any-time answers. That is, we want to quickly maintain up-to-date answers as soon as we observe a new slice matrix $\mathbf{S}^{(t)}$. Some representative examples of our methods are also shown in Fig. 1.

3 Dynamic Proximity and Centrality: Definitions

In this section, we introduce our proximity and centrality definitions for dynamic bipartite graphs. We begin by reviewing random walk with restart, which is a good proximity measurement for static graphs. We then extend it to the dynamic setting by 1) using different ways to aggregate edges from different time steps, that is to place different emphasis on more recent links; and 2) using *degree-preservation* to achieve monotonicity for dynamic proximity.

3.1 Background: Static Setting

Among many others, one very successful method to measure proximity is random walk with restart (RWR), which has been receiving increasing interest in recent years—see Section 7 for a detailed review.

For a static bipartite graph, random walk with restart is defined as follows: Consider a random particle that starts from node i. The particle iteratively transits to its neighbors with probability proportional to the corresponding edge weights. Also at each step, the particle returns to node i with some restart probability $(1 - c)$. The proximity score from node i to node j is defined as the steady-state probability $r_{i,j}$ that the particle will be on node j [24]. Intuitively, $r_{i,j}$ is the fraction of time that the particle starting from node i will spend on each node j of the graph, after an infinite number of steps.

If we represent the bipartite graph as a uni-partite graph with the following square adjacency matrix $\mathbf{W}$ and degree matrix $\mathbf{D}$:

$$\mathbf{W} = \begin{pmatrix} \mathbf{0}_{n\times n} & \mathbf{M} \\ \mathbf{M}' & \mathbf{0}_{l\times l} \end{pmatrix}$$

$$\mathbf{D} = \begin{pmatrix} \mathbf{D}_1 & \mathbf{0}_{n\times l} \\ \mathbf{0}_{l\times n} & \mathbf{D}_2 \end{pmatrix} \quad (3.1)$$

then, all the proximity scores $r_{i,j}$ between all possible node pairs i, j are determined by the matrix $\mathbf{Q}$:

$$\begin{aligned} r_{i,j} &= \mathbf{Q}(i,j) \\ \mathbf{Q} &= (1-c)\cdot(\mathbf{I}_{(n+l)\times(n+l)} - c\mathbf{D}^{-1}\mathbf{W})^{-1} \end{aligned} \quad (3.2)$$

Based on the dynamic proximity as in equation 3.4, we define the centrality for a given source node s as the average proximity score from all nodes in the graph (including s itself) to s. For simplicity, we ignore the time step superscript. That is,

$$\text{centrality}(s) \triangleq \frac{\sum_{i=1}^{n+l} r_{i,s}}{n+l} \quad (3.3)$$

3.2 Dynamic Proximity

Since centrality is defined in terms of proximity, we will henceforth focus only on the latter. In order to apply the random walk with restart (see equation 3.2) to the dynamic setting, we need to address two subtle but important points.

The first is how to update the adjacency matrix $\mathbf{M}^{(t)}$, based on the observed slice matrix $\mathbf{S}^{(t)}$. As mentioned before, usually it is not enough to consider only the current

slice matrix $\mathbf{S}^{(t)}$. For example, examining publications from conferences in a single year may lead to proximity scores that vary widely and reflect more "transient" effects (such as a bad year for an author), rather than "true" shifts in his affinity to research areas (for example, a shift of interest from databases to data mining, or a change of institutions and collaborators). Similarly, examining movie ratings from a single day may not be sufficient to accurately capture the proximity of, say, two users in terms of their tastes. Thus, in subsection 3.2.1, we propose three different strategies to aggregate slices into an adjacency matrix $\mathbf{M}^{(t)}$ or, equivalently, to update $\mathbf{M}^{(t)}$. Note, however, that single-slice analysis can be viewed as a special case of the "sliding window" aggregation strategy.

The second point is related to the "monotonicity" of proximity versus time. In a dynamic setting with only link additions and weight increases (i.e., $\mathbf{S}^{(t)}(i,j) \geq 0$, for all time steps t and nodes i, j), in many applications it is desirable that the proximity between any two nodes does not drop. For example, consider an author-conference bipartite graph, where edge weights represent the number of papers that an author has published in the corresponding conference. We would like a proximity measure that represents the total contribution/credit that an author has accumulated in each conference. Intuitively, this score should not decrease over time. In subsection 3.2.2, we propose *degree-preservation* to achieve this property.

3.2.1 Updating the adjacency matrix.

As explained above, it is usually desirable to analyze multiple slices together, placing different emphasis on links based on their age. For completeness, we describe three possible aggregation schemes.

Global Aggregation. The first way to obtain the adjacency matrix $\mathbf{M}^{(t)}$ is to simply add the new edges or edge weights in $\mathbf{S}^{(t)}$ to the previous adjacency matrix $\mathbf{M}^{(t-1)}$ as follows:

$$\mathbf{M}^{(t)} = \sum_{j=1}^{t} \mathbf{S}^{(j)}$$

We call this scheme *global aggregation*. It places equal emphasis on all edges from the beginning of time and, only in this case, $\Delta\mathbf{M}^{(t)} = \mathbf{S}^{(t)}$. Next, we define schemes that place more emphasis on recent links. For both of these schemes, $\Delta\mathbf{M}^{(t)} \neq \mathbf{S}^{(t)}$.

Sliding Window. In this case, we only consider the edges and weights that arrive in the past len time steps, where the parameter len is the length of the sliding window:

$$\mathbf{M}^{(t)} = \sum_{j=\max\{1,\ t-len+1\}}^{t} \mathbf{S}^{(j)}$$

Exponential Weighting. In this case, we "amplify" the new edges and weights at time t by an exponential factor $\beta^j (\beta > 1)$: $\mathbf{M}^{(t)} = \sum_{j=1}^{t} \beta^j \mathbf{S}^{(j)}$.

3.2.2 Fixed degree matrix.

In a dynamic setting, if we apply the actual degree matrix $\mathbf{D}^{(t)}$ to equation (3.2) at time t, the monotonicity property will not hold. To address this issue, we propose to use degree-preservation [17, 31]. That is, we use the same degree matrix $\tilde{\mathbf{D}}$ at all time steps.

Thus, our proximity $r_{i,j}^{(t)}$ from node i to node j at time step t is formally defined as in equation (3.4). The adjacency matrix $\mathbf{M}^{(t)}$ is computed by any update method in subsection 3.2 and the fixed degree matrix $\tilde{\mathbf{D}}$ is set to be a constant (a) times the degree matrix at the first time step—we always set $a = 1000$ in this paper.

$$\begin{aligned} r_{i,j}^{(t)} &= \mathbf{Q}^{(t)}(i,j) \\ \mathbf{Q}^{(t)} &= (1-c) \cdot (\mathbf{I}_{(n+l)\times(n+l)} - c\tilde{\mathbf{D}}^{-1}\mathbf{W}^{(t)})^{-1} \\ \mathbf{W}^{(t)} &= \begin{pmatrix} \mathbf{0}_{n\times n} & \mathbf{M}^{(t)} \\ \mathbf{M}'^{(t)} & \mathbf{0}_{l\times l} \end{pmatrix} \\ \tilde{\mathbf{D}} &= a \cdot \mathbf{D}^{(1)} \end{aligned} \tag{3.4}$$

We have the following lemma for our dynamic proximity (equation (3.4)). By the lemma 3.1, if the actual degree $\mathbf{D}^{(t)}(i,i)$ does not exceed the fixed degree $\tilde{\mathbf{D}}(i,i)$ (condition 2), then the proximity between any two nodes will never drop as long as the edge weights in adjacency matrix $\mathbf{M}^{(t)}$ do not drop (condition 1).

LEMMA 3.1. **Monotonicity Property of Dynamic Proximity** *If (1) all elements in the difference matrix $\Delta\mathbf{M}^{(t)}$ are non-negative; and (2) $\mathbf{D}^{(t)}(i,i) \leq \tilde{\mathbf{D}}(i,i)$ $(i = 1, 2, ..., (n+l))$; then we have $r_{i,j}^{(t)} \geq r_{i,j}^{(t-1)}$ for any two nodes (i,j).*

Proof: First of all, since $\mathbf{D}^{(t)}(i,i) \leq \tilde{\mathbf{D}}(i,i)$, we have $\|c\tilde{\mathbf{D}}^{-1}\mathbf{W}^{(t)}\|^k \rightarrow 0$ as $k \rightarrow \infty$. Therefore, we have $\mathbf{Q}^{(t)} = (1-c)\sum_{k=0}^{\infty}(c\tilde{\mathbf{D}}^{-1}\mathbf{W}^{(t)})^k$. On the other hand, since all elements in the difference matrix $\Delta\mathbf{M}^{(t)}$ are non-negative, we have $\mathbf{W}^{(t)}(i,j) \geq \mathbf{W}^{(t-1)}(i,j)$ for any two nodes (i,j). Therefore, we have $\mathbf{Q}^{(t)}(i,j) \geq \mathbf{Q}^{(t-1)}(i,j)$ for any two nodes (i,j), which completes the proof. □

Finally, we should point out that a, $\mathbf{D}$ and the non-negativity of $\mathbf{M}$ are relevant only if a monotonic score is desired. Even without these assumptions, the correctness or efficiency of our proposed algorithms are not affected. If non-monotonic scores are permissible, none of these assumptions are necessary.

4 Dynamic Proximity: Computations

4.1 Preliminaries: BB_LIN on Static Graphs

In this section, we introduce our fast solutions to efficiently track dynamic proximity. We will start with BB_LIN [32], a fast algorithm for static, skewed bipartite graphs. We then extend it to the dynamic setting.

One problem with random walk with restart is computational efficiency, especially for large graphs. According to the definition (equation (3.4)), we need to invert an $(n+l) \times (n+l)$ matrix. This operation is prohibitively

Algorithm 1 BB_LIN

Input: The adjacency matrix at time t, as in equation (3.1); and the query nodes i and j.
Output: The proximity $r_{i,j}$ from node i to node j.
1: **Pre-Computation Stage(Off-Line):**
2: normalize for type 1 objects: $\mathbf{Mr} = \mathbf{D}_1^{-1} \cdot \mathbf{M}$
3: normalize for type 2 objects: $\mathbf{Mc} = \mathbf{D}_2^{-1} \cdot \mathbf{M}'$
4: compute the core matrix: $\mathbf{\Lambda} = (\mathbf{I} - c^2\mathbf{Mc} \cdot \mathbf{Mr})^{-1}$
5: store the matrices: $\mathbf{Mr}$, $\mathbf{Mc}$, and $\mathbf{\Lambda}$.
6: **Query Stage (On-Line):**
7: **Return:** $r_{i,j} = \textbf{GetQij}(\mathbf{\Lambda}, \mathbf{Mr}, \mathbf{Mc}, i, j, c)$

slow for large graphs. In [32], the authors show that for skewed, static bipartite graphs, we only need to pre-compute and store a matrix inversion of size $l \times l$ to get all possible proximity scores. BB_LIN, which is the starting point for our fast algorithms, is summarized in Alg. 1.

Algorithm 2 GetQij

Input: The core matrix $\mathbf{\Lambda}$, the normalized adjacency matrices $\mathbf{Mr}$ (for type 1 objects), and $\mathbf{Mc}$ (for type 2), and the query nodes i and j $(1 \le i, j \le (n + l))$.
Output: The proximity $r_{i,j}$ from node i to node j
1: **if** $i \le n$ and $j \le n$ **then**
2: $q(i,j) = 1(i = j) + c^2\mathbf{Mr}(i,:) \cdot \mathbf{\Lambda} \cdot \mathbf{Mc}(:,j)$
3: **else if** $i \le n$ and $j > n$ **then**
4: $q(i,j) = c\mathbf{Mr}(i,:) \cdot \mathbf{\Lambda}(:,j-n)$
5: **else if** $i > n$ and $j \le n$ **then**
6: $q(i,j) = c\mathbf{\Lambda}(i-n,:) \cdot \mathbf{Mc}(:,j)$
7: **else**
8: $q(i,j) = \mathbf{\Lambda}(i-n, j-n)$
9: **end if**
10: **Return:** $r_{i,j} = (1-c)q(i,j)$

Based on Alg. 1, we only need to pre-compute and store a matrix inversion $\mathbf{\Lambda}$ of size $l \times l$. For skewed bipartite graphs ($l \ll m, n$), $\mathbf{\Lambda}$ is much cheaper to pre-compute and store. For example, on the entire **NetFlix** user-movie bipartite graph, which contains about $2.7M$ users, about $18K$ movies and more than $100M$ edges (see Section 6 for the detailed description of the dataset), it takes 1.5 hours to pre-compute the $18K \times 18K$ matrix inversion $\mathbf{\Lambda}$. For pre-computation stage, this is quite acceptable.

On the other hand, in the on-line query stage, we can get any proximity scores using the function **GetQij**[2] . This stage is also cheap in terms of computation. For example, to output a proximity score between two type-1 objects (step 2 in **GetQij**) , only one sparse vector-matrix multiplication and one vector-vector multiplication are needed. For a proximity score between one type-1 object and one type-2 object, only one sparse vector-vector multiplication (step 4 and step 6) is necessary. Finally, for a proximity score between two type-2 objects (step 8), only retrieving one element in the matrix $\mathbf{\Lambda}$ is needed. As an example, on the **NetFlix** dataset, it takes less than 1 second to get one proximity score. Note that all possible proximity scores are determined by the matrix $\mathbf{\Lambda}$ (together with the normalized adjacency matrices $\mathbf{Mr}$ and $\mathbf{Mc}$). We thus refer to the matrix $\mathbf{\Lambda}$ as the the *core matrix*.

[2]Note that in step 2 of **GetQij**, $1(.)$ is the indicator function, i.e. it is 1 if the condition in (.) is true and 0 otherwise.

4.2 Challenges for Dynamic Setting

In a dynamic setting, since the adjacency matrix changes over time, the core matrix $\mathbf{\Lambda}^{(t)}$ is no longer constant. In other words, the steps 1-4 in Alg. 1 themselves become a part of the on-line stage since we need to update the core matrix $\mathbf{\Lambda}^{(t)}$ at each time step. If we still rely on the straightforward strategy (i.e., the steps 1-4 in Alg. 1) to update the core matrix (referred to as "Straight-Update"), the total computational complexity for each time step is $O(l^3 + m \cdot l)$. Such complexity is undesirable for the online stage. For example, 1.5 hours to recompute the core matrix for the **NetFlix** dataset is unacceptably long.

Thus, our goal is to efficiently update the core matrix $\mathbf{\Lambda}^{(t)}$ at time step t, based on the previous core matrix $\mathbf{\Lambda}^{(t-1)}$ and the difference matrix $\Delta\mathbf{M}^{(t)}$. For simplicity, we shall henceforth assume the use of the global aggregation scheme to update the adjacency matrix. However, the ideas can be easily applied to the other schemes, sliding window and exponential weighting.

4.3 Our Solution 1: Single Update

Next, we describe a fast algorithm (*Fast-Single-Update*) to update the core matrix $\mathbf{\Lambda}^{(t)}$ at time step t, if only one edge (i_0, j_0) changes at time t. In other words, there is only one non-zero element in $\Delta\mathbf{M}^{(t)}$: $\Delta\mathbf{M}^{(t)}(i_0, j_0) = w_0$. To simplify the description of our algorithm, we present the difference matrix $\Delta\mathbf{M}^{(t)}$ as a from-to list: $[i_0, j_0, w_0]$.

Algorithm 3 *Fast-Single-Update*

Input: The core matrix $\mathbf{\Lambda}^{(t-1)}$, the normalized adjacency matrices $\mathbf{Mr}^{(t-1)}$ (for type 1 objects) and $\mathbf{Mc}^{(t-1)}$ (for type 2 objects) at time step $t-1$, and the difference list $[i_0, j_0, w_0]$ at the time step t.
Output: The core matrix $\mathbf{\Lambda}^{(t)}$, the normalized adjacency matrices $\mathbf{Mr}^{(t)}$ and $\mathbf{Mc}^{(t)}$ at time step t.
1: $\mathbf{Mr}^{(t)} = \mathbf{Mr}^{(t-1)}$, and $\mathbf{Mc}^{(t)} = \mathbf{Mc}^{(t-1)}$.
2: $\mathbf{Mr}^{(t)}(i_0, j_0) = \mathbf{Mr}^{(t)}(i_0, j_0) + \frac{w_0}{\tilde{\mathbf{D}}(i_0, i_0)}$
3: $\mathbf{Mc}^{(t)}(j_0, i_0) = \mathbf{Mc}^{(t)}(j_0, i_0) + \frac{w_0}{\tilde{\mathbf{D}}(j_0+n, j_0+n)}$
4: $\mathbf{X} = \mathbf{0}_{l\times 2}$, and $\mathbf{Y} = \mathbf{0}_{2\times l}$
5: $\mathbf{X}(:,1) = \mathbf{Mc}^{(t)}(:, i_0)$, and $\mathbf{X}(j_0, 2) = \frac{w_0}{\tilde{\mathbf{D}}(j_0+n, j_0+n)}$
6: $\mathbf{Y}(1, j_0) = \frac{c^2 \cdot w_0}{\tilde{\mathbf{D}}(i_0, i_0)}$, and $\mathbf{Y}(2,:) = c^2 \cdot \mathbf{Mr}^{(t-1)}(i_0,:)$
7: $\mathbf{L} = (\mathbf{I}_{2\times 2} - \mathbf{Y} \cdot \mathbf{\Lambda}^{(t-1)} \cdot \mathbf{X})^{-1}$
8: $\mathbf{\Lambda}^{(t)} = \mathbf{\Lambda}^{(t-1)} + \mathbf{\Lambda}^{(t-1)} \cdot \mathbf{X} \cdot \mathbf{L} \cdot \mathbf{Y} \cdot \mathbf{\Lambda}^{(t-1)}$

The correctness of *Fast-Single-Update* is guaranteed by the following theorem:

THEOREM 4.1. **Correctness of *Fast-Single-Update*.** *The matrix* $\mathbf{\Lambda}^{(t)}$ *maintained by* Fast-Single-Update *is exactly the core matrix at time step t, i.e.,* $\mathbf{\Lambda}^{(t)} = (\mathbf{I} - c^2\mathbf{Mc}^{(t)}\mathbf{Mr}^{(t)})^{-1}$.

Proof: first of all, since only one edge (i_0, j_0) is updated at time t, only the i_0^{th} row of the matrix $\mathbf{Mr}^{(t)}$ and the i_0^{th} column of the matrix $\mathbf{Mc}^{(t)}$ change at time t

Let $\mathbf{V}^{(t)} = c^2\mathbf{Mc}^{(t)} \cdot \mathbf{Mr}^{(t)}$, and $\mathbf{V}^{(t-1)} = c^2\mathbf{Mc}^{(t-1)} \cdot \mathbf{Mr}^{(t-1)}$. By the spectral representation of $\mathbf{V}^{(t)}$ and $\mathbf{V}^{(t-1)}$, we have the following equation:

$$\begin{aligned} \mathbf{V}^t &= c^2 \sum_{k=1}^{n} \mathbf{Mc}^{(t)}(:,k) \cdot \mathbf{Mr}^{(t)}(k,:) \\ &= \mathbf{V}^{t-1} + \delta \end{aligned} \tag{4.5}$$

where δ indicates the difference between $\mathbf{V}^{(t)}$ and $\mathbf{V}^{(t-1)}$. This gives us:

$$\delta = \sum_{s=0}^{1} (-1)^s \cdot c^2\mathbf{Mc}^{(t)}(:,i_0) \cdot \mathbf{Mr}^{(t-s)}(i_0,:) = \mathbf{X} \cdot \mathbf{Y}$$

where the matrices $\mathbf{X}$ and $\mathbf{Y}$ are defined in steps 4-6 of Alg. 3. Putting all the above together, we have

$$\mathbf{\Lambda}^t = (\mathbf{I} - \mathbf{V}^t)^{-1} = (\mathbf{I} - \mathbf{V}^{t-1} - \mathbf{X} \cdot \mathbf{Y})^{-1} \tag{4.6}$$

Applying the Sherman-Morrison Lemma [25] to equation (4.6), we have

$$\mathbf{\Lambda}^{(t)} = \mathbf{\Lambda}^{(t-1)} + \mathbf{\Lambda}^{(t-1)} \cdot \mathbf{X} \cdot \mathbf{L} \cdot \mathbf{Y} \cdot \mathbf{\Lambda}^{(t-1)}$$

where the 2×2 matrix $\mathbf{L}$ is defined in step 7 of Alg. 3. This completes the proof. □

Fast-Single-Update is significantly more computationally efficient, as shown by the next lemma. In particular, the complexity of *Fast-Single-Update* is only $O(l^2)$, as opposed to $O(l^3 + ml)$ for the straightforward method.

LEMMA 4.1. **Efficiency of *Fast-Single-Update*.** *The computational complexity of Fast-Single-Update is* $O(l^2)$.

Proof: Omitted for space. □

4.4 Our Solutions 2: Batch Update

In many real applications, more than one edges typically change at each time step. In other words, there are multiple non-zero elements in the difference matrix $\Delta\mathbf{M}^{(t)}$. Suppose we have a total of $\hat{m}$ edge changes at time step t. An obvious choice is to repeatedly call *Fast-Single-Update* $\hat{m}$ times.

An important observation from many real applications is that it is unlikely these $\hat{m}$ edges are randomly distributed. Instead, they typically form a low-rank structure. That is, if these $\hat{m}$ edges involve $\hat{n}$ type 1 objects and $\hat{l}$ type 2 objects, we have $\hat{n} \ll \hat{m}$ or $\hat{l} \ll \hat{m}$. For example, in an author-conference bipartite graph, we will often add a group of $\hat{m}$ new records into the database at one time step. In most cases, these new records only involve a small number of authors and/or conferences—see Section 6 for the details. In this section, we show that we can do a single batch update (*Fast-Batch-Update*) on the core matrix. This is much more efficient than either doing $\hat{m}$ single updates repeatedly, or recomputing the core matrix from scratch. The main advantage of our approach lies on the observation that the difference matrix has low rank, and our upcoming algorithm needs time proportional to the *rank*, as opposed to the number of changed edges $\hat{m}$. This holds in real settings, because when a node is modified, several of its edges are changed (e.g., an author publishes several papers in a given conferences each year).

Let $\mathcal{I} = \{i_1, ..., i_{\hat{n}}\}$ be the indices of the involved type 1 objects. Similarly, let $\mathcal{J} = \{j_1, ..., j_{\hat{l}}\}$ be the indices of the involved type 2 objects. We can represent the difference matrix $\Delta\mathbf{M}^{(t)}$ as an $\hat{n} \times \hat{l}$ matrix. In order to simplify the description of the algorithm, we define two matrices $\Delta\mathbf{M}r$ and $\Delta\mathbf{M}c$ as follows:

$$\begin{aligned} \Delta\mathbf{M}r(k,s) &= \frac{\Delta\mathbf{M}^{(t)}(i_k, j_s)}{\tilde{\mathbf{D}}(i_k, i_k)} \\ \Delta\mathbf{M}c(s,k) &= \frac{\Delta\mathbf{M}^{(t)}(j_s, i_k)}{\tilde{\mathbf{D}}(j_s + n, j_s + n)} \\ (k = 1, ..., \hat{n}, s &= 1, ..., \hat{l}) \end{aligned} \tag{4.7}$$

The correctness of *Fast-Batch-Update* is guaranteed by the following theorem:

THEOREM 4.2. **Delta Matrix Inversion Theorem.** *The matrix* $\mathbf{\Lambda}^{(t)}$ *maintained by* Fast-Batch-Update *is exactly the core matrix at time step t, i.e.,* $\mathbf{\Lambda}^{(t)} = (\mathbf{I} - c^2\mathbf{Mc}^{(t)}\mathbf{Mr}^{(t)})^{-1}$.

Proof: Similar to *Fast-Single-Update*. Omitted for space. □

The efficiency of *Fast-Single-Update* is given by the following lemma. Note that the linear term $O(\hat{m})$ comes from equation (4.7), since we need to scan the non-zero elements of the difference matrix $\Delta\mathbf{M}^{(t)}$. Compared to the straightforward recomputation which is $O(l^3 + ml)$, *Fast-Batch-Update* is $O(\min(\hat{\mathrm{l}}, \hat{\mathrm{n}}) \cdot \mathrm{l}^2 + \hat{\mathrm{m}})$. Since $\min(\hat{\mathrm{l}}, \hat{\mathrm{n}}) < \mathrm{l}$ always holds, as long as we have $\hat{m} < m$, *Fast-Single-Update* is always more efficient. On the other hand, if we do $\hat{m}$ repeated single updates using *Fast-Single-Update*, the computational complexity is $O(\hat{m}l^2)$. Thus, since typically $\min(\hat{\mathrm{l}}, \hat{\mathrm{n}}) \ll \hat{\mathrm{m}}$, *Fast-Batch-Update* is much more efficient in this case.

LEMMA 4.2. **Efficiency of *Fast-Batch-Update*.** *The computational complexity of Fast-Batch-Update is* $O(\min(\hat{\mathrm{l}}, \hat{\mathrm{n}}) \cdot \mathrm{l}^2 + \hat{\mathrm{m}})$.

Proof: Omitted for space. □

Algorithm 4 *Fast-Batch-Update*

Input: The core matrix $\mathbf{\Lambda}^{(t-1)}$, the normalized adjacency matrices $\mathbf{Mr}^{(t-1)}$ (for type 1 objects) and $\mathbf{Mc}^{(t-1)}$ (for type 2 objects) at time step $t-1$, and the difference matrix $\Delta\mathbf{M}^{(t)}$ at the time step t

Output: The core matrix $\mathbf{\Lambda}^{(t)}$, the normalized adjacency matrices $\mathbf{Mr}^{(t)}$ and $\mathbf{Mc}^{(t)}$ at time step t.

1: $\mathbf{Mr}^{(t)} = \mathbf{Mr}^{(t-1)}$, and $\mathbf{Mc}^{(t)} = \mathbf{Mc}^{(t-1)}$.
2: define $\Delta\mathbf{M}r$ and $\Delta\mathbf{M}c$ as in equation (4.7)
3: $\mathbf{Mr}^{(t)}(\mathcal{I},\mathcal{J}) = \mathbf{Mr}^{(t)}(\mathcal{I},\mathcal{J}) + \Delta\mathbf{M}r$
4: $\mathbf{Mc}^{(t)}(\mathcal{J},\mathcal{I}) = \mathbf{Mc}^{(t)}(\mathcal{J},\mathcal{I}) + \Delta\mathbf{M}c$
5: let $\hat{k} = \min(\hat{\text{l}}, \hat{\text{n}})$. let $\mathbf{X} = \mathbf{0}_{l\times 2\hat{k}}$, and $\mathbf{Y} = \mathbf{0}_{2\hat{k}\times l}$
6: **if** $\hat{l} < \hat{n}$ **then**
7: $\quad\mathbf{X}(:, 1:\hat{l}) = \mathbf{Mc}^{(t-1)}(:,\mathcal{I})\cdot\Delta\mathbf{M}r$
8: $\quad\mathbf{Y}(\hat{l}+1:2\hat{l},:) = \Delta\mathbf{M}c\cdot\mathbf{Mr}^{(t-1)}(\mathcal{I},:)$
9: $\quad\mathbf{X}(\mathcal{J},1:\hat{l}) = \mathbf{X}(\mathcal{J},1:\hat{l}) + \Delta\mathbf{M}c\cdot\Delta\mathbf{M}r$
10: $\quad\mathbf{X}(\mathcal{J},1:\hat{l}) = \mathbf{X}(\mathcal{J},1:\hat{l}) + \mathbf{Y}(\hat{l}+1:2\hat{l},\mathcal{J})$
11: $\quad\mathbf{Y}(\hat{l}+1:2\hat{l},\mathcal{J}) = 0$
12: $\quad$**for** $k = 1:\hat{k}$ **do**
13: $\quad\quad$set $\mathbf{Y}(k, j_k) = 1$, and $\mathbf{X}(j_k, k+\hat{k}) = 1$
14: $\quad$**end for**
15: $\quad$set $\mathbf{X} = c^2\cdot\mathbf{X}$, and $\mathbf{Y} = c^2\cdot\mathbf{Y}$
16: **else**
17: $\quad\mathbf{X}(:,1:\hat{n}) = \mathbf{Mc}^{(t)}(:,\mathcal{I})$
18: $\quad\mathbf{X}(\mathcal{J},\hat{n}+1:2\hat{n}) = \Delta\mathbf{M}c$
19: $\quad\mathbf{Y}(1:\hat{n},\mathcal{J}) = c^2\cdot\Delta\mathbf{M}r$
20: $\quad\mathbf{Y}(\hat{n}+1:2\hat{n},:) = c^2\cdot\mathbf{Mr}^{(t-1)}(\mathcal{I},:)$
21: **end if**
22: $\mathbf{L} = (\mathbf{I}_{2\hat{k}\times 2\hat{k}} - \mathbf{Y}\cdot\mathbf{\Lambda}^{(t-1)}\cdot\mathbf{X})^{-1}$
23: $\mathbf{\Lambda}^{(t)} = \mathbf{\Lambda}^{(t-1)} + \mathbf{\Lambda}^{(t-1)}\cdot\mathbf{X}\cdot\mathbf{L}\cdot\mathbf{Y}\cdot\mathbf{\Lambda}^{(t-1)}$

5 Dynamic Proximity: Applications

In this section, we give the complete algorithms for the two applications we posed in Section 2, that is, *Track-Centrality* and *Track-Proximity*. For each case, we can track top-k queries over time. For *Track-Centrality*, we can also track the centrality (or the centrality rank) for an individual node. For *Track-Proximity*, we can also track the proximity (or the proximity rank) for a given pair of nodes.

In all the cases, we first need the following function (i.e., Alg. 5) to do initialization. Then, at each time step, we update (i) the normalized adjacency matrices, $\mathbf{Mc}^{(t)}$ and $\mathbf{Mr}^{(t)}$, as well as the core matrix, $\mathbf{\Lambda}^{(t)}$; and we perform (ii) one or two sparse matrix-vector multiplications to get the proper answers. Compared to the update time (part (i)), the running time for part (ii) is always much less. So our algorithms can quickly give the proper answers at each time step. On the other hand, we can easily verify that our algorithms give the exact answers, without any quality loss or approximation.

Algorithm 5 Initialization

Input: The adjacency matrix at time step 1 $\mathbf{M}^{(1)}$, and the parameter c.

Output: The fixed degree matrix $\tilde{\mathbf{D}}$, the normalized matrices at time step 1 $\mathbf{Mr}^{(1)}$ and $\mathbf{Mc}^{(1)}$, and the initial core matrix $\mathbf{\Lambda}^{(1)}$.

1: get the fixed degree matrix $\tilde{\mathbf{D}}$ as equation (3.4)
2: normalize for type 1 objects: $\mathbf{Mr}^{(1)} = \mathbf{D}_1^{-1}\cdot\mathbf{M}^{(1)}$
3: normalize for type 2 objects: $\mathbf{Mc}^{(1)} = \mathbf{D}_2^{-1}\cdot\mathbf{M}'^{(1)}$
4: get the core matrix: $\mathbf{\Lambda}^{(1)} = (\mathbf{I} - c^2\mathbf{Mc}^{(1)}\cdot\mathbf{Mr}^{(1)})^{-1}$
5: store the matrices: $\mathbf{Mr}^{(1)}$, $\mathbf{Mc}^{(1)}$, and $\mathbf{\Lambda}^{(1)}$.

5.1 *Track-Centrality*

Here, we want to track the top-k most important type 1 (and/or type 2) nodes over time. For example, on an author-conference bipartite graph, we want to track the top-10 most influential authors (and/or conferences) over time. For a given query node, we also want to track its centrality (or the rank of centrality) over time. For example, on an author-conference bipartite graph, we can track the relative importance of an author in the entire community.

Based on the definition of centrality (equation 3.3) and the fast update algorithms we developed in Section 4, we can get the following algorithm (Alg. 6) to track the top-k queries over time. The algorithm for tracking centrality for a single query node is quite similar to Alg. 6. We omit the details for space.

Algorithm 6 *Track-Centrality* (Top-k Queries)

Input: The time-evolving bipartite graphs $\{\mathbf{M}^{(1)}, \Delta\mathbf{M}^{(t)}(t\ge 2)\}$, the parameters c and k

Output: The top-k most central type 1 (and type 2) objects at each time step t.

1: **Initialization**
2: **for** each time step $t (t\ge 1)$ **do**
3: $\quad x = \mathbf{1}_{1\times n}\cdot\mathbf{Mr}^{(t)}\cdot\mathbf{\Lambda}^{(\mathbf{t})}$; and $y = \mathbf{1}_{1\times l}\cdot\mathbf{\Lambda}^{(\mathbf{t})}$
4: $\quad\vec{r_2}' = c\cdot x + y$
5: $\quad\vec{r_1}' = c\cdot\vec{r_2}\cdot\mathbf{Mc}^{(t)}$
6: $\quad$output the top k type 1 objects according to $\vec{r_1}'$ (larger value means more central)
7: $\quad$output the top k type 2 objects according to $\vec{r_2}'$ (larger value means more central)
8: $\quad$Update $\mathbf{Mr}^{(t)}$, $\mathbf{Mc}^{(t)}$, and $\mathbf{\Lambda}^{(\mathbf{t})}$ for $t\ge 2$.
9: **end for**

In step 8 of Alg. 6, we can either use *Fast-Single-Update* or *Fast-Batch-Update* to update the normalized matrices $\mathbf{Mr}^{(t)}$ and $\mathbf{Mc}^{(t)}$, and the core matrix $\mathbf{\Lambda}^{(\mathbf{t})}$. The running time for steps 3–8 is much less than the update time (step 8). Thus, *Track-Centrality* can give the ranking results quickly at each time step. On the other hand, using elementary linear algebra, we can easily prove the correctness of *Track-*

Centrality:

LEMMA 5.1. **Correctness of *Track-Centrality*.** *The vectors $\vec{r_1}'$ and $\vec{r_2}'$ in Alg. 6 provide a correct ranking of type 1 and type 2 objects at each time step t. That is, the ranking is exactly according to the centrality defined in equation* (3.3)*.*

5.2 *Track-Proximity*

Here, we want to track the top-k most related/relevant type 1 (and/or type 2) objects for object A at each time step. For example, on an author-conference bipartite graph evolving over time, we want track "*Which are the major conferences for John Smith in the past 5 year?*" or "*Who are most the related authors for John Smith so far?*" For a given pair of nodes, we also want to track their pairwise relationship over time. For example, in an author-conference bipartite graph evolving over time, we can track "*How much credit (a.k.a proximity) John Smith has accumulated in KDD?*"

The algorithm for top-k queries is summarized in Alg. 7. The algorithm for tracking the proximity for a given pair of nodes is quite similar to Alg. 7. We omit its details for space.

In Alg. 7, again, at each time step, the update time will dominate the total computational time. Thus by using either *Fast-Single-Update* or *Fast-Batch-Update*, we can quickly give the ranking results at each time step. Similar to *Track-Proximity*, we have the following lemma for the correctness of *Track-Proximity*:

LEMMA 5.2. **Correctness of *Track-Proximity*.** *The vectors $\vec{r_1}'$ and $\vec{r_2}'$ in Alg. 7 provide a correct ranking of type 1 and type 2 objects at each time step t. That is, the ranking is exactly according to the proximity defined in* (3.4)*.*

Algorithm 7 *Track-Proximity* (Top-*k* Queries)

Input: The time-evolving bipartite graphs $\{\mathbf{M}^{(1)}, \Delta\mathbf{M}^{(t)}(t \geq 2)\}$, the parameters c and k, and the source node s.

Output: The top-k most related type 1 (and type 2) objects for s at each time step t.

1: **Initialization**
2: **for** each time step $t(t \geq 1)$ **do**
3:   **for** $i = 1 : n$ **do**
4:     $r_{s,i} = \mathbf{GetQij}(\mathbf{\Lambda}^{(t)}, \mathbf{Mr}^{(t)}, \mathbf{Mc}^{(t)}, s, i, c))$
5:   **end for**
6:   let $\vec{r_1} = [r_{s,i}](i = 1, ...n)$
7:   **for** $j = 1 : l$ **do**
8:     $r_{s,j} = \mathbf{GetQij}(\mathbf{\Lambda}^{(t)}, \mathbf{Mr}^{(t)}, \mathbf{Mc}^{(t)}, s, j + n, c))$
9:   **end for**
10:   let $\vec{r_2} = [r_{s,j}](j = 1, ...l)$
11:   output the top *k* type 1 objects according to $\vec{r_1}'$ (larger value means more relevant)
12:   output the top *k* type 2 objects according to $\vec{r_2}'$ (larger value means more relevant)
13:   update $\mathbf{Mr}^{(t)}$, $\mathbf{Mc}^{(t)}$, and $\mathbf{\Lambda^{(t)}}$ for $t \geq 2$.
14: **end for**

6 Experimental Results

In this section we present experimental results, after we introduce the datasets in subsection 6.1. All the experiments are designed to answer the following questions:

- *Effectiveness:* What is the quality of the applications (*Track-Centrality* and *Track-Proximity*) we proposed in this paper?
- *Efficiency:* How fast are the proposed algorithms (*Fast-Single-Update* and *Fast-Batch-Update*for the update time, *Track-Centrality* and *Track-Proximity* for the overall running time)?

6.1 Datasets.

We use five different datasets in our experiments, summarized in Table 6.1. We verify the effectiveness of our proposed dynamic centrality measures on **NIPS**, **DM**, and **AC**, and measure the efficiency of our algorithms using the larger **ACPost** and **NetFlix** datasets.

The first dataset (**NIPS**) is from the NIPS proceedings[3]. The timestamps are publication years, so each graph slice $\mathbf{M}$ corresponds to one year, from 1987 to 1999. For each year, we have an author-paper bipartite graph. Rows represent authors and columns represent papers. Unweighted edges between authors and papers represent authorship. There are 2,037 authors, 1,740 papers, and 13 time steps (years) in total with an average of 308 new edges per year.

The **DM**, **AC**, and **ACPost** datasets are from DBLP[4]. For the first two, we use paper publication years as timestamps, similar to **NIPS**. Thus each graph slice $\mathbf{S}$ corresponds to one year.

DM uses author-paper information for each year between 1995–2007, from a restricted set of conferences, namely the five major data mining conferences ('KDD', 'ICDM', 'SDM', 'PKDD', and 'PAKDD'). Similar to **NIPS**, rows represent authors, columns represent papers and unweighted edges between them represent authorship. There are 5,095 authors, 3,548 papers, and 13 time steps (years) in total, with an average of 765 new edges per time step.

AC uses author-conference information from the entire DBLP collection, between years 1959–2007. In contrast to **DM**, columns represent conferences and edges connect authors to conferences they have published in. Each edge in $\mathbf{S}$ is weighted by the number of papers published by the author in the corresponding conference for that year. There are 418,236 authors, 3,571 conferences, and 49 time steps (years) with an average of 26,508 new edges at each time step.

[3] http://www.cs.toronto.edu/~roweis/data.html

[4] http://www.informatik.uni-trier.de/~ley/db/

Table 2: Datasets used in evaluations

Name	$n \times l$	Ave.$\hat{m}$	time steps
NIPS	2,037×1,740	308	13
DM	5,095× 3,548	765	13
AC	418,236×3,571	26,508	49
ACPost	418,236×3,571	1,007	1258
NetFlix	2,649,429×17,770	100,480,507	NA

ACPost is primarily used to evaluate the scalability of our algorithms. In order to obtain a larger number of timestamps at a finer granularity, we use posting date on DBLP (the 'mdate' field in the XML archive of DBLP, which represents when the paper was entered into the database), rather than publication year. Thus, each graph slice **S** corresponds to one day, between 2002-01-03 and 2007-08-24. **ACPost** is otherwise similar to **AC**, with number of papers as edge weights. There are 418,236 authors, 3,571 conferences, and 1,258 time steps (days with at least one addition into DBLP), with an average of 1,007 new edges per day.

The final dataset, **NetFlix**, is from the Netflix prize[5]. Rows represent users and columns represent movies. If a user has rated a particular movie, we connect them with an unweighted edge. This dataset consists of one slice and we use it in subsection 6.2 to evaluate the efficiency *Fast-Single-Update*. In total, we have 2,649,429 users, 17,770 movies, and 100,480,507 edges.

6.2 Effectiveness: Case Studies

Here, we show the experimental results for the three applications on real datasets, all of which are consistent with our intuition.

6.2.1 Results on *Track-Centrality*.

We apply Alg. 6 to the **NIPS** dataset. We use "Global Aggregation" to update the adjacency matrix $\mathbf{M}^{(t)}$. We track the top-k ($k = 10$) most central (i.e.influential) authors in the whole community. Table 3 lists the results for every two years. The results make sense: famous authors in the NIPS community show up in the top-10 list and their relative rankings change over time, reflecting their activity/influence in the whole NIPS community up to that year. For example, Prof. Terrence J. Sejnowski ('Sejnowski_T') shows in the top-10 list from 1989 on and his ranking keeps going up in the following years (1991,1993). He remains number 1 from 1993 on. Sejnowski is one of the founders of NIPS, an IEEE Fellow, and the head of the Computational Neurobiology Lab at the Salk institute. The rest of the top-placed researchers include Prof. Michael I. Jordan ('Jordan_M') from UC Berkeley and Prof. Geoffrey E. Hinton ('Hinton_G') from Univ. of Toronto, well known for their work in graphical models and neural networks, respectively. We can also track the centrality values as well as their rank for an individual author over the years. Fig. 1(a) plots the centrality ranking for some authors over the years. Again, the results are consistent with intuition. For example, Michael I. Jordon starts to have significant influence (within top-30) in the NIPS community from 1991 on; his influence rapidly increases in the following up years (1992-1995); and stays within the top-3 from 1996 on. Prof. Christof Koch ('Koch_C') from Caltech remains one of the most influential (within top-3) authors in the whole NIPS community over the years (1990-1999).

[5] http://www.netflixprize.com/

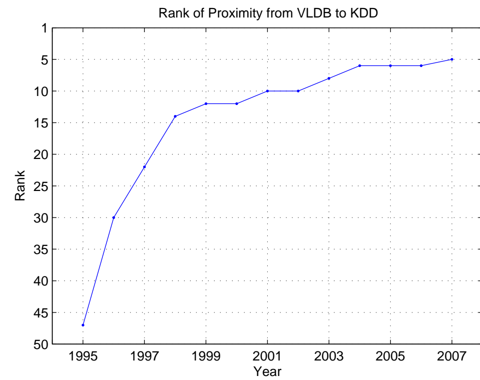


Figure 2: The rank of the proximity from 'VLDB' to 'KDD' up to each year

6.2.2 Results on *Track-Proximity*.

We first report the results on the **DM** dataset. We use "Global Aggregation" to update the adjacency matrix at each time step. In this setting, we can track the top-k most related papers/authors in the data mining community for a given query author up to each year. Table. 4 lists the top-5 most related authors for 'Jian Pei' over the years (2001-2007). The results make perfect sense: (1) the top co-author (Prof. 'Jiawei Han') is Prof. Jian Pei's advisor; (2) the other top collaborators are either from SUNY-Buffalo (Prof. Aidong Zhang), or from IBM-Watson (Drs. Philip S. Yu, Haixun Wang, Wei Wang), which is also reasonable, since Prof. Pei held a faculty position at SUNY-Buffalo; (3) the IBM-Watson collaboration ('Philip S. Yu' and 'Haixun Wang') got stronger over time.

Then, we apply *Track-Proximity* on the dataset **AC**. Here, we want to track the proximity ranking for a given pair of nodes over time. Fig. 2 plots the rank of proximity from the 'VLDB' conference to the 'KDD' conference. We use "Global Aggregation" to update the adjacency matrix. In this way, proximity between the 'VLDB' and 'KDD' conferences measures the importance/relevance of 'KDD' wrt 'VLDB' up to each year. From the figure, we can see that the rank of 'KDD' keeps going up, reaching the fifth position by 2007. The other top-4 conferences for 'VLDB' by 2007 are 'SIGMOD', 'ICDE', 'PODS' and 'EDBT', in this order. The result makes sense: with more and more multi-disciplinary authors publishing in both communities

Table 3: Top-10 most influential (central) authors up to each year.

1987	1989	1991	1993	1995	1997	1999
'Abbott_L'	'Bower_J'	'Hinton_G'	'Sejnowski_T'	'Sejnowski_T'	'Sejnowski_T'	'Sejnowski_T'
'Burr_D'	'Hinton_G'	'Koch_C'	'Koch_C'	'Jordan_M'	'Jordan_M'	'Koch_C'
'Denker_J'	'Tesauro_G'	'Bower_J'	'Hinton_G'	'Hinton_G'	'Koch_C'	'Jordan_M'
'Scofield_C	'Denker_J'	'Sejnowski_T'	'Mozer_M'	'Koch_C'	'Hinton_G'	'Hinton_G'
'Bower_J'	'Mead_C'	'LeCun_Y'	'LeCun_Y'	'Mozer_M'	'Mozer_M'	'Mozer_M'
'Brown_N'	'Tenorio_M'	'Mozer_M'	'Denker_J'	'Bengio_Y'	'Dayan_P'	'Dayan_P'
'Carley_L'	'Sejnowski_T'	'Denker_J'	'Bower_J'	'Lippmann_R'	'Bengio_Y'	'Singh_S'
'Chou_P'	'Lippmann_R'	'Waibel_A'	'Kawato_M'	'LeCun_Y'	'Barto_A'	'Bengio_Y'
'Chover_J'	'Touretzky_D'	'Moody_J'	'Waibel_A'	'Waibel_A'	'Tresp_V'	'Tresp_V'
'Eeckman_F'	'Koch_C'	'Lippmann_R'	'Simard_P'	'Simard_P'	'Moody_J'	'Moody_J'

Table 4: Top-5 most related authors for 'Jian Pei' up to each year.

2001	2003	2005	2007
'Jiawei_Han'	'Jiawei_Han'	'Jiawei_Han'	'Jiawei_Han'
'Behzad_Mortazavi-Asl'	'Behzad_Mortazavi-Asl'	'Haixun_Wang'	'Haixun_Wang'
'Hongjun_Lu'	'Aidong_Zhang'	'Aidong_Zhang'	'Philip_S._Yu'
'Meichun_Hsu'	'Philip_S._Yu'	'Philip_S._Yu'	'Wei_Wang'
'Shiwei_Tang'	'Hongjun_Lu'	'Wei_Wang'	'Aidong_Zhang'

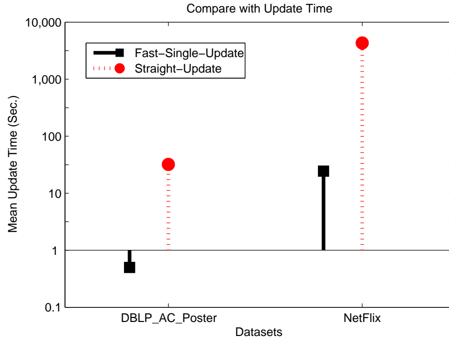


Figure 3: Evaluation of *Fast-Single-Update*. For both datasets, one edge changes at each time step. The running time is averaged over multiple runs of experiments and shown in logarithmic scale.

(databases and data mining), 'KDD' becomes more and more closely related to 'VLDB'.

We also test the top-k queries on **AC**. Here, we use "Sliding Window" (with window length $\mathrm{len} = 5$) to update the adjacency matrix. In this setting, we want to track the top-k most related conferences/authors for a given query node in the past 5 years at each time step t. Fig. 1(b) lists the top-5 conferences for Dr. 'Philip S. Yu'. The major research interest (top-5 conferences) for 'Philip S. Yu' is changing over time. For example, in the years 1988-1992, his major interest is in databases ('ICDE' and 'VLDB'), performance ('SIGMETRICS') and distributed systems ('ICDCS' and 'PDIS'). However, during 2003-2007, while databases ('ICDE' and 'VLDB') are still one of his major research interests, data mining became a strong research focus ('KDD', 'SDM' and 'ICDM').

6.3 Efficiency

After initialization, at each time step, most time is spent on updating the core matrix $\mathbf{\Lambda}^{(t)}$, as well as the normalized adjacency matrices. In this subsection, we first report the running time for update and then give the total running time for each time step. We use the two largest datasets (**ACPost** and **NetFlix**) to measure performance.

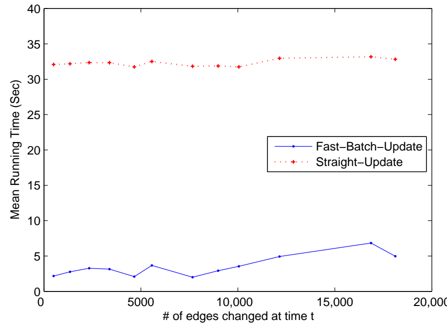
6.3.1 Update Time

We first evaluate *Fast-Single-Update*. Both **ACPost** and **NetFlix** are used. For each dataset, we randomly add one new edge into the graph and compute the update time. The experiments are run multiple times. We compare *Fast-Single-Update* with Straight-Update (which does $l \times l$ matrix inversion at each time step) and the result is summarized in Fig. 3—Note that the y-axis is in log-scale). On both datasets, *Fast-Single-Update* requires significantly less computation: on **ACPost**, it is 64x faster (0.5 seconds vs. 32 seconds), while on **NetFlix**, it is 176x faster (22.5 seconds vs $4,313$ seconds).

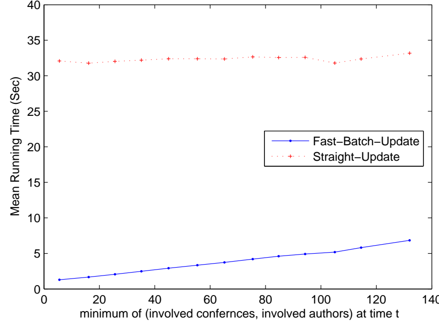
To evaluate *Fast-Batch-Update*, we use **ACPost**. From $t = 2$ and on, at each time step, we have between $\hat{m} = 1$ and $\hat{m} = 18,121$ edges updated. On average, there are 913 edges updated at each time step t ($t \geq 2$). Note that despite the large number of updated edges for some time steps, the rank of the difference matrix (i.e. $\min(\hat{n}, \hat{l})$) at each time step is relatively small, ranging from 1 to 132 with an average of 33. The results are summarized in Fig 4. We plot the mean update time vs. the number ($\hat{m}$) of changed edges in Fig 4(a) and the mean update time vs. the rank ($\min(\hat{n}, \hat{l})$) of the update matrix in Fig 4(b). Compared to the Straight-Update, *Fast-Batch-Update* is again much faster, achieving 5–32x speed-up. On average, it is 15x faster than Straight-Update.

6.3.2 Total Running Time

Here, we study the total running time at each time step for *Track-Centrality*. The results for *Track-Proximity* are similar and omitted for space. For *Track-Centrality*, we let the algorithm return both the top-10 type 1 objects and the top-10 type 2 objects. We use the **NetFlix** dataset with one



(a) Running Time vs. $\hat{m}$



(b) Running Time vs. $\min(\hat{n}, \hat{l})$

Figure 4: Evaluation on *Fast-Batch-Update*.

edge changed at each time step and **ACPost** dataset with multiple edges changed at each time step.

We compare our algorithms ("*Track-Centrality*") with (i) the one that uses Straight-Update in our algorithms (still referred as "Straight-Update"); and (ii) that uses iterative procedure [27] to compute proximity and centrality at each time step (referred as 'Ite-Alg"). The results are summarized in Fig. 5. We can see that in either case, our algorithm (*Track-Centrality*) is much faster. For example, it takes 27.8 seconds on average on the **NetFlix** dataset, which is 155x faster over"Straight-Update" (4,315 seconds); and is 93x faster over "Ite-Alg" (2,582 seconds). In either case, the update time for *Track-Centrality* dominates the overall running time. For example, on the **ACPost** dataset, update time accounts for more than 90% of the overall running time (2.4 seconds vs. 2.6 seconds). Thus, when we have to track queries for many nodes of interest, the advantage of *Track-Centrality* over "Ite-Alg" will be even more significant, since at each time step we only need to do update once for all queries, while the running time of "Ite-Alg" will increase linearly with respect to the number of queries.

7 Related Work

In this section, we review the related work, which can be categorized into two parts: static graph mining and dynamic graph mining.

Static Graph Mining. There is a lot of research work on static graph mining, including pattern and law mining [2,

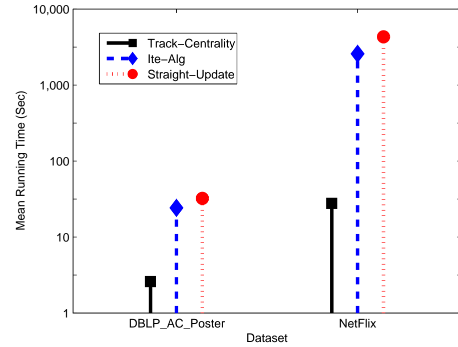


Figure 5: Overall running time at each time step for *Track-Centrality*. For **ACPost**, there are multiple edges changed at each time step; and for **NetFlix**, there is only one edge changed at each time step. The running time is averaged in multiple runs of experiments and it is in the logarithm scale

7, 9, 5, 22], frequent substructure discovery [33], influence propagation [16], and community mining [10][12][13].

In terms of centrality, Google's PageRank algorithm [23] is the most related. The proposed *Track-Centrality* can actually be viewed as its generalization for dynamic bipartite graphs. As for proximity, the closest work is random walk with restart [15, 24, 32]. The proposed *Track-Proximity* is its generalization for dynamic bipartite graphs. Other representative proximity measurements on static graphs include the sink-augmented delivered current [8], cycle free effective conductance [17], survivable network [14], and direction-aware proximity [31]. Although we focus on random walk with restart in this paper, our fast algorithms can be easily adapted to other random walk based measurements, such as [8, 31]. Also, there are a lot of applications of proximity measurements. Representative work includes connection subgraphs [8, 17, 29], neighborhood formation in bipartite graphs [27], content-based image retrieval [15], cross-modal correlation discovery [24], the BANKS system [1], link prediction [20], pattern matching [30], detecting anomalous nodes and links in a graph [27], ObjectRank [4] and RelationalRank [11].

Dynamic Graph Mining. More recently, there is an increasing interest in mining time-evolving graphs, such as densification laws and shrinking diameters [19], community evolution [3], dynamic tensor analysis [28], and dynamic communities [6, 26]. To the best of our knowledge, there is no previous work on proximity for time-evolving graphs. Remotely related work in the sparse literature on the topic is [21]. However, we have a different setting and focus compared with [21]: we aim to incrementally track the proximity and centrality for nodes of interest by quickly updating the core matrix (as well as the adjacency matrices), while in [21] the authors focus on efficiently using time information by adding time as explicit nodes in the graph.

8 Conclusion

In this paper, we study how to incrementally track the node proximity as well as the centrality for time-evolving bipartite graphs. To the best of our knowledge, we are the first to study the node proximity and centrality in this setting. The major contributions of the paper include:

1: Proximity and centrality definitions for time-evolving graphs.

2: Two fast update algorithms (*Fast-Single-Update* and *Fast-Batch-Update*), that do not resort to approximation and hence guarantee no quality loss (see Theorem 4.2).

3: Two algorithms to incrementally track centrality (*Track-Centrality*) and proximity (*Track-Proximity*), in any-time fashion.

4: Extensive experimental case-studies on several real datasets, showing how different queries can be answered, achieving up to **15∼176x** speed-up.

We can achieve such speedups while providing exact answers because we carefully leverage the fact that the rank of graph updates is small, compared to the size of the original matrix. Our experiments on real data show that this typically translates to at least an order of magnitude speedup.

References

[1] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, and S. S. Parag. Banks: Browsing and keyword searching in relational databases. In *VLDB*, pages 1083–1086, 2002.

[2] R. Albert, H. Jeong, and A.-L. Barabasi. Diameter of the world wide web. *Nature*, (401):130–131, 1999.

[3] L. Backstrom, D. P. Huttenlocher, J. M. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, pages 44–54, 2006.

[4] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, pages 564–575, 2004.

[5] A. Broder, R. Kumar, F. Maghoul1, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web: experiments and models. In *WWW Conf.*, 2000.

[6] Y. Chi, X. Song, D. Zhou, K. Hino, and B. L. Tseng. Evolutionary spectral clustering by incorporating temporal smoothness. In *KDD*, pages 153–162, 2007.

[7] S. Dorogovtsev and J. Mendes. Evolution of networks. *Advances in Physics*, 51:1079–1187, 2002.

[8] C. Faloutsos, K. S. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *KDD*, pages 118–127, 2004.

[9] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *SIGCOMM*, pages 251–262, Aug-Sept. 1999.

[10] G. Flake, S. Lawrence, C. L. Giles, and F. Coetzee. Self-organization and identification of web communities. *IEEE Computer*, 35(3), Mar. 2002.

[11] F. Geerts, H. Mannila, and E. Terzi. Relational link-based ranking. In *VLDB*, pages 552–563, 2004.

[12] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *9th ACM Conf. on Hypertext and Hypermedia*, pages 225–234, New York, 1998.

[13] M. Girvan and M. E. J. Newman. Community structure is social and biological networks.

[14] M. Grötschel, C. L. Monma, and M. Stoer. Design of survivable networks. In *Handbooks in Operations Research and Management Science 7: Network Models*. North Holland, 1993.

[15] J. He, M. Li, H.-J. Zhang, H. Tong, and C. Zhang. Manifold-ranking based image retrieval. In *ACM Multimedia*, pages 9–16, 2004.

[16] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing the spread of influence through a social network. *KDD*, 2003.

[17] Y. Koren, S. C. North, and C. Volinsky. Measuring and extracting proximity in networks. In *KDD*, 2006.

[18] D. C. Kozen. *The Design and Analysis Algorithms*. Springer-Verlag, 1992.

[19] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.

[20] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. In *Proc. CIKM*, 2003.

[21] E. Minkov and W. W. Cohen. An email and meeting assistant using graph walks. In *CEAS*, 2006.

[22] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.

[23] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998. Paper SIDL-WP-1999-0120 (version of 11/11/1999).

[24] J.-Y. Pan, H.-J. Yang, C. Faloutsos, and P. Duygulu. Automatic multimedia cross-modal correlation discovery. In *KDD*, pages 653–658, 2004.

[25] W. Piegorsch and G. E. Casella. Inverting a sum of matrices. In *SIAM Review*, volume 32, pages 470–470, 1990.

[26] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu. Graphscope: parameter-free mining of large time-evolving graphs. In *KDD*, pages 687–696, 2007.

[27] J. Sun, H. Qu, D. Chakrabarti, and C. Faloutsos. Neighborhood formation and anomaly detection in bipartite graphs. In *ICDM*, pages 418–425, 2005.

[28] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *KDD*, pages 374–383, 2006.

[29] H. Tong and C. Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *KDD*, pages 404–413, 2006.

[30] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, pages 737–746, 2007.

[31] H. Tong, C. Faloutsos, and Y. Koren. Fast direction-aware proximity for graph mining. In *KDD*, pages 747–756, 2007.

[32] H. Tong, C. Faloutsos, and J.-Y. Pan. Random walk with restart: Fast solutions and applications. *Knowledge and Information Systems: An International Journal (KAIS)*, 2007.

[33] D. Xin, J. Han, X. Yan, and H. Cheng. Mining compressed frequent-pattern sets. In *VLDB*, pages 709–720, 2005.