Random Walk with Restart over Dynamic Graphs

Weiren Yu [♮,†] and Julie McCann [†]

[†] *Department of Computing, Imperial College London*
[♮] *School of Engineering & Applied Science, Aston University*
`w.yu3@aston.ac.uk` `j.mccann@imperial.ac.uk`

***Abstract*—Random Walk with Restart (RWR) is an appealing measure of proximity between nodes based on graph structures. Since real graphs are often large and subject to minor changes, it is prohibitively expensive to recompute proximities from scratch. Previous methods use LU decomposition and degree reordering heuristics, entailing $O(|V|^3)$ time and $O(|V|^2)$ memory to compute all ($|V|^2$) pairs of node proximities in a static graph. In this paper, a dynamic scheme to assess RWR proximities is proposed:**

(1) For unit update, we characterize the changes to all-pairs proximities as the outer product of two vectors. We notice that the multiplication of an RWR matrix and its transition matrix, unlike traditional matrix multiplications, is commutative. This can greatly reduce the computation of all-pairs proximities from $O(|V|^3)$ to $O(|\Delta|)$ time for each update without loss of accuracy, where $|\Delta|$ ($\ll |V|^2$) is the number of affected proximities.

(2) To avoid $O(|V|^2)$ memory for all pairs of outputs, we also devise efficient partitioning techniques for our dynamic model, which can compute all pairs of proximities segment-wisely within $O(l|V|)$ memory and $O(\lceil \frac{|V|}{l} \rceil)$ I/O costs, where $1 \le l \le |V|$ is a user-controlled trade-off between memory and I/O costs.

(3) For bulk updates, we also devise aggregation and hashing methods, which can discard many unnecessary updates further and handle chunks of unit updates simultaneously.

Our experimental results on various datasets demonstrate that our methods can be 1–2 orders of magnitude faster than other competitors while securing scalability and exactness.

I. Introduction

With the increasing scale of the Internet, many applications are dealing with dynamically evolving graphs on a large scale. For example, the World Wide Web today embraces more than a trillion links, and 7%–18% of them are updated fortnightly. A key task to manage graphs is link-based proximity search, *i.e.*, given a graph $G = (V, E)$ with $|V|$ nodes and $|E|$ edges, the retrieval of all proximities between every two nodes in G.

Recently, Random Walk with Restart (RWR) [11] has been proposed as an attractive proximity measure. The success of RWR is mainly ascribed to its intuitive concept that revolves around random walks. Consider a random surfer starting from a given node x. The surfer has two options at every step — either moving to one of its out-links, or restarting from x with a certain probability. After the stability is iteratively attained, *the RWR proximity* of every node v *w.r.t.* a given node x is the steady-state probability that the surfer will eventually arrive at node v. RWR has a wide spectrum of applications in *e.g.*, nearest neighbor search [13], named entity disambiguation [6], collaborative filtering [5], and automatic image labeling [11].

Compared with other measures [15]–[19], RWR has three salient features: (1) It can recursively capture both direct and indirect neighboring information of each node. (2) Unlike

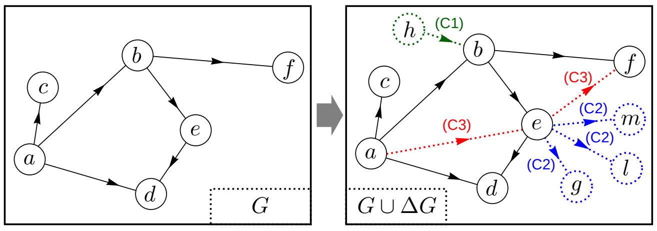


Figure 1: A Website Updated by Three Types of Edge Insertions

SimRank [4] which is a symmetric measure to quantify the structural equivalence between two nodes, RWR is an asymmetric measure in a digraph, with the focus on reachability from one node to another. (3) RWR is a stable measure that is resilient to noise in a graph. With these advantages, RWR is very popular in fertile communities [5], [6], [9]–[13].

However, the practicability of RWR is hindered by its high computational cost. The best-known methods [3], [8] exploit LU decomposition with degree reordering heuristics, which requires $O(|V|^3)$ time and $O(|V|^2)$ memory in the worst case to evaluate all ($|V|^2$) pairs of proximities over a static graph. Due to the dynamics and growing size of the Internet, it is too expensive to recompute all pairs of proximities from scratch when a graph is frequently updated with small changes.

Motivated by this, we consider efficient dynamic computation of all pairs of RWR proximities on large evolving graphs. Given all-pairs proximities in old graph G, and updates ΔG to G (*i.e.*, a collection of new edge insertions or deletions), our goal is to evaluate only the changes to all pairs of RWR proximities efficiently without loss of exactness. Particularly, we are interested in the situation: When all-pairs proximities cannot fit into memory, can our dynamic RWR model compute such changes over each segment independently while securing high efficiency? Let us take the following example.

Example 1. *Figure 1 is a part of the school website G, where each node is a web page, and each edge is a hyperlink.*

In this semester, the website is updated by adding 4 new pages and 6 new links (see the dashed edges, denoted as ΔG). To update all-pairs proximities in $G \cup \Delta G$, instead of using a batch method to reassess all new proximities from scratch, one can incrementally retrieve only the changes to the old proximities in response to graph updates ΔG to G, by reusing the information of the old proximity matrix in G.

However, due to memory limitations, the entire old proximity matrix may not fit in memory. Thus, it is highly desirable that, in our dynamical RWR model, each segment of the old proximity matrix can be updated independently (in parallel). □

Algorithm	Time (All Pairs)	Memory	I/Os	Error
Inc-R	$O(\|I_{\max}\|\|\Delta\|)$	$O(l\|V\|)$	$O(\|I_{\max}\|\lceil\frac{\|V\|}{l}\rceil)$	0
Inc-uR	$O(\|\Delta G\|\|\Delta\|)$	$O(l\|V\|)$	$O(\|\Delta G\|\lceil\frac{\|V\|}{l}\rceil)$	0
Bear [8]	$O(\|\Delta G\|\|E\|\|V\|)$	$O(\|V\|^2)$	0	0
k-dash [3]	$O(\|\Delta G\|\|V\|^3)$	$O(\|V\|^2)$	0	0
MC [2]	$O(\frac{\log(\|\Delta G\|)}{(1-\gamma)^2}\|V\|^2)$	$O(\|V\|^2)$	0	Prob.
DAP [10]	$O(K\|\Delta G\|\|E\|\|V\|)$	$O(\|V\|^2)$	0	γ^{K+1}
B-LIN [11]	$O(\frac{1}{\tau^2}\|\Delta G\|\|V\|^3)$	$O(\|V\|^2+\nu\|V\|)$	0	$\epsilon_{\text{rank-}\nu}$

Table I: Compare Inc-R with others for $|\Delta G|$ edge updates, where $|I_{\max}| \ll |\Delta G|$, $|\Delta| \ll |V|^2$, τ is the partition number, K is the number of iterations, ν is the target rank of SVD, and l is a user-controlled tradeoff between memory and I/Os

Due to the recursive nature of RWR definition, there are two grand challenges to dynamic RWR computation: (1) It seems difficult to exploit the relationship among proximity changes, old proximities in G, and ΔG. (2) It is hard to avoid $O(|V|^2)$ memory to update all ($|V|^2$) pairs of proximities.

Contributions. To address the above challenges, we propose a novel dynamic scheme, Inc-R, with three main ingredients:

(1) For unit update, we devise an efficient dynamic model that can characterize the changes to all-pairs RWR as the outer product of two vectors. Moreover, we notice that the multiplication of an RWR proximity matrix and its transition matrix, unlike traditional matrix multiplication, is commutative. These can substantially reduce the time to compute all ($|V|^2$) pairs of proximities from $O(|V|^3)$ to $O(|V|^2)$ in the worst case for each update without loss of accuracy. In general, the $O(|V|^2)$ time can be reduced to $O(|\Delta|)$ further, where $|\Delta|$ ($\leq |V|^2$) is the number of affected RWR elements. (Section III)

(2) To reduce its memory further, we also propose efficient partitioning techniques, in which all ($|V|^2$) pairs proximities can be updated segment-wisely in just $O(l|V|)$ memory with $O(\lceil\frac{|V|}{l}\rceil)$ I/O costs, where $1 \leq l \leq |V|$ is a user-controlled trade-off between memory and I/O costs. (Section IV)

(3) For bulk updates, we propose aggregation and hashing techniques for pure updates (*i.e.*, either insertions or deletions are allowable) and mixture updates. These can minimize many unnecessary updates further and handle chunks of unit updates simultaneously. (Section V)

Related Work. In Table I, we summarize the complexity of the state-of-the-art methods for all-pairs RWR computation.

Tong *et al.* [11] provided a pioneering SVD-based method, B-LIN. It first splits a graph into τ dense blocks and a sparse block, and then performs matrix inverse for each dense block and a rank-ν SVD approximation for the sparse block.

Recently, Fujiwara *et al.* [3] devised an LU decomposition method, k-dash, for top-K RWR search *w.r.t.* a given query. k-dash entails $O(|V|^2)$ time and $O(|V|^2)$ memory to retrieve only the top-K proximities in one column of an RWR matrix. Thus, it yields $O(|V|^3)$ time for top-K all-pairs RWR search. In contrast, our incremental approach can compute all pairs of proximities accurately in $O(|V|^2)$ worst-case time.

The existing work [2], [7] provided probabilistic methods to estimate proximities. Sarkar *et al.* [7] integrated a sampling approach with branch and bound pruning. Bahmani *et al.* [2] used Monte Carlo to incrementally estimate proximities.

Zhu *et al.* [20] used a hub length-based scheduling scheme to prioritize random walks for RWR approximation. It differs from our work in that our method is exact and needs only two "pivot proximity vectors" to describe affected regions.

The recent work of [14] proposed an incremental method to compute RWR. However, this method would fail when either end node of an inserted edge is a new one. In contrast, our techniques can overcome this limitation.

Recently, Shin *et al.* [8] have given a fresh impetus to fast RWR computation on static graphs. Their scheme, Bear, combines a block elimination approach with a Schur complement of submatrix derived from the LU decomposition.

II. Preliminaries

In this section, we briefly overview the background of RWR. For graph $G = (V, E)$, let $\mathcal{O}(j)$ be the out-degree of node j. Let $\mathbf{A}$ be the backward transition matrix defined as

$$\mathbf{A}_{i,j} = 1/\mathcal{O}(j) \text{ if } \exists (j,i) \in E, \text{ and } \mathbf{A}_{i,j} = 0 \text{ otherwise.}$$

Then, RWR proximity matrix $\mathbf{P} \in \mathbb{R}^{|V|\times|V|}$ is defined by

$$\mathbf{P} = \gamma \mathbf{A}\mathbf{P} + (1-\gamma)\mathbf{I} \tag{1}$$

where $\mathbf{I} \in \mathbb{R}^{|V|\times|V|}$ is an identity matrix, and $(1-\gamma) \in (0,1)$ is *restarting probability* that is often set to 0.1 ($\gamma = 0.9$) [11].

In vector forms, Eq.(1) can also be rewritten as

$$\mathbf{P}_{\star,x} = \gamma \mathbf{A}\mathbf{P}_{\star,x} + (1-\gamma)\mathbf{e}_x \qquad (\forall x \in V) \tag{2}$$

where $\mathbf{P}_{\star,x}$ is the x-th column of $\mathbf{P}$, denoting the proximities of all nodes *w.r.t.* node x; and $\mathbf{e}_x$ is the x-th column of $\mathbf{I}$.

III. Unit Update

We mainly focus on *unit insertion* (Sections III-A–III-E). Similar techniques also apply to *unit deletion* (Section III-F).

Note that all methods in this section are *in-memory* based, *i.e.*, all-pairs old and new proximities need fit in main memory. In Section IV, we will extend our methods to handle the cases when all-pairs proximities cannot fit in main memory.

Given graph $G = (V, E)$, for edge (i, j) to be added to G, we consider four cases in each subsection, respectively:

(C1) $i \notin V$ and $j \in V$; (C2) $i \in V$ and $j \notin V$;
(C3) $i \in V$ and $j \in V$; (C4) $i \notin V$ and $j \notin V$.

Example 2. *Figure 1 depicts an old digraph G (in solid lines) with a set of edge insertions ΔG (dash lines) into G. In ΔG, the insertion (h, b) belongs to case (C1); $(e, g), (e, l), (e, m)$ to case (C2); $(a, e), (e, f)$ to case (C3).* □

A. Inserting Edge (i, j) with $i \notin V$ and $j \in V$

We first consider the case (C1): the insertion of edge (i, j) with $i \notin V$ and $j \in V$. After insertion, $\tilde{\mathbf{A}}$ becomes

$$\tilde{\mathbf{A}} = \begin{array}{c} \begin{matrix} \overbrace{\phantom{\mathbf{A}}}^{|V| \text{ cols}} & \overbrace{\phantom{\mathbf{e}_j}}^{\text{col } i} \end{matrix} \\ \left[\begin{matrix} \mathbf{A} & \mathbf{e}_j \\ \mathbf{0} & 0 \end{matrix}\right] \end{array} \begin{matrix} \} \ |V| \text{ rows} \\ \leftarrow \text{row } i \end{matrix} \tag{3}$$

Note that in Eq.(3), though the last row of $\tilde{\mathbf{A}}$ is $\mathbf{0}$, we need not replace $\mathbf{0}$ by $\frac{1}{|V|+1}\mathbf{1}^T$ (where $\mathbf{1}^T$ is a row vector of all 1s). This is because, unlike the existence of PageRank vector that requires $\tilde{\mathbf{A}}$ irreduciblity, the existence of RWR is ensured by $(\mathbf{I} - \gamma\tilde{\mathbf{A}})$ invertibility. Since $(\mathbf{I} - \gamma\tilde{\mathbf{A}})$ is diagonally dominant, it is invertible, and thereby its RWR matrix $\tilde{\mathbf{P}}$ exists.

To derive new $\tilde{\mathbf{P}}$ from old $\mathbf{P}$, we have the following result.

Theorem 1. *Given a graph $G = (V, E)$ and an old proximity matrix $\mathbf{P}$, after edge (i, j) with $i \notin V$ and $j \in V$ is inserted, the new proximity matrix $\tilde{\mathbf{P}}$ can be computed as*

$$\tilde{\mathbf{P}} = \left[\begin{array}{c|c} \mathbf{P} & \gamma \mathbf{P}_{\star,j} \\ \hline \mathbf{0} & 1-\gamma \end{array}\right] \begin{array}{l} \} \ |V| \textit{ rows} \\ \leftarrow \textit{row } i \end{array}$$

Theorem 1 provides an efficient way to dynamically obtain new $\tilde{\mathbf{P}}$ from old $\mathbf{P}$ for insertion case (C1). Specifically, new $\tilde{\mathbf{P}}$ is a $(|V|+1) \times (|V|+1)$ matrix formed by bordering old $\mathbf{P}$ by 3 parts: (a) column vector $\gamma \mathbf{P}_{\star,j}$, (b) row vector $\mathbf{0}$, and (c) scalar $(1-\gamma)$. Thus, it requires $O(|V|)$ time to incrementally compute $\tilde{\mathbf{P}}$, which is dominated by the complexity of $\gamma \mathbf{P}_{\star,j}$.

Notice that the upper-right block $\gamma \mathbf{P}_{\star,j}$ is a scalar multiple of $\mathbf{P}_{\star,j}$. This is because, after edge $(i, j)_{i \notin V, j \in V}$ is inserted, the random walks from i to node $x \in V$ are a concatenation of the edge $i \to j$ and the old random walks from j to x:

$$\underbrace{\overbrace{i \Rightarrow j}^{\text{new edge}} \overbrace{\to \circ \to \cdots \to \cdots \to \cdots \circ \to x}^{\text{old walks from } j \text{ to } x \text{ tallied by } \mathbf{P}_{x,j}}}_{\text{new walks from } i \text{ to } x \text{ tallied by } \tilde{\mathbf{P}}_{x,i}}$$

Since the out-degree of node i is 1, the old walks from j to x (tallied by $\mathbf{P}_{x,j}$) can be reused with just a multiple factor to evaluate the new walks from i to x (tallied by $\tilde{\mathbf{P}}_{x,i}$).

Example 3. *Recall the old graph G in Figure 1 (left). Given $\gamma = 0.9$ and old proximity matrix $\mathbf{P}$ for G, when edge (h, b) is inserted, new $\tilde{\mathbf{P}}$ can be updated via Theorem 1 as*

$$\tilde{\mathbf{P}} = \begin{array}{c|cccccc|c} & a & b & c & d & e & f & h \\ a & 0.100 & 0 & 0 & 0 & 0 & 0 & 0 \\ b & 0.030 & 0.100 & 0 & 0 & 0 & 0 & 0.090 \\ c & 0.030 & 0 & 0.100 & 0 & 0 & 0 & 0 \\ d & 0.042 & 0.041 & 0 & 0.100 & 0.090 & 0 & 0.037 \\ e & 0.014 & 0.045 & 0 & 0 & 0.100 & 0 & 0.041 \\ f & 0.014 & 0.045 & 0 & 0 & 0 & 0.100 & 0.041 \\ \hline h & 0 & 0 & 0 & 0 & 0 & 0 & 0.100 \end{array}$$

old proximity matrix $\mathbf{P}$

$\gamma \mathbf{P}_{\star,b}$

$\times\gamma$

zero row vector $\mathbf{0}$

$1-\gamma$

B. Inserting Edge (i, j) with $i \in V$ and $j \notin V$

We next consider the case (C2): the insertion of edge (i, j) with $i \in V$ and $j \notin V$. This case is more difficult than (C1) since this type of insertion will change not only the dimension of old transition matrix $\mathbf{A}$, but also a number of entries of $\mathbf{A}$.

Lemma 1. *Given old graph $G = (V, E)$ and its old transition matrix $\mathbf{A}$. After edge (i, j) with $i \in V$ and $j \notin V$ is inserted, the new transition matrix $\tilde{\mathbf{A}}$ becomes*

$$\tilde{\mathbf{A}} = \begin{bmatrix} \overbrace{\mathbf{A}}^{|V| \textit{ cols}} & \overbrace{\mathbf{0}}^{\textit{col } j} \\ \mathbf{e}_i^T & 0 \end{bmatrix} \begin{array}{l} \} \ |V| \textit{ rows} \\ \leftarrow \textit{row } j \end{array} \qquad \textit{if } \mathcal{O}(i) = 0; \tag{4}$$

$$\tilde{\mathbf{A}} = \begin{bmatrix} \overbrace{\mathbf{A} + \mathbf{v}\mathbf{e}_i^T}^{|V| \textit{ cols}} & \overbrace{\mathbf{0}}^{\textit{col } j} \\ \frac{1}{\mathcal{O}(i)+1} \mathbf{e}_i^T & 0 \end{bmatrix} \begin{array}{l} \} \ |V| \textit{ rows} \\ \leftarrow \textit{row } j \end{array} \qquad \textit{if } \mathcal{O}(i) \neq 0, \tag{5}$$

where $\mathbf{v} = -\frac{1}{\mathcal{O}(i)+1} \mathbf{A}_{\star,i} \in \mathbb{R}^{|V| \times 1}$, and $\mathcal{O}(i)$ denotes the out-degree of node i in old graph G.

Utilizing the structure of $\tilde{\mathbf{A}}$, we next propose an efficient technique that can incrementally update new $\tilde{\mathbf{P}}$ from old $\mathbf{P}$. Our main idea is to convert the computation of $\tilde{\mathbf{P}}$ into solving $(\mathbf{I} - \gamma \tilde{\mathbf{A}})^{-1}$ in terms of old $\mathbf{P}$. When we combine Lemma 1 with the block matrix inverse formula, new $\tilde{\mathbf{P}}$ will become

$$\tilde{\mathbf{P}} = (1-\gamma) \begin{bmatrix} \left(\mathbf{I} - \gamma \mathbf{A} + \mathbf{A}_{\star,i} \mathbf{y}^T\right)^{-1} & \mathbf{0} \\ \mathbf{y}^T \left(\mathbf{I} - \gamma \mathbf{A} + \mathbf{A}_{\star,i} \mathbf{y}^T\right)^{-1} & 1 \end{bmatrix} \begin{array}{l} \} \ |V| \text{ rows} \\ \leftarrow \text{row } j \end{array}$$

This structure suggests that, once $\mathbf{y}$ is determined, solving $\tilde{\mathbf{P}}$ can boil down to solving $\left(\mathbf{I} - \gamma \mathbf{A} + \mathbf{A}_{\star,i} \mathbf{y}^T\right)^{-1}$ in terms of $\mathbf{P}$. Fortunately, it is unnecessary to obtain $\left(\mathbf{I} - \gamma \mathbf{A} + \mathbf{A}_{\star,i} \mathbf{y}^T\right)^{-1}$ from scratch because this inverse can be computed efficiently from $(\mathbf{I} - \gamma \mathbf{A})^{-1}$ perturbed by the rank-one update $\mathbf{A}_{\star,i} \mathbf{y}^T$. However, as $(\mathbf{I} - \gamma \mathbf{A})^{-1}$ can be expressed as scaling of $\mathbf{P}$, the challenge is that: *Can we describe the changes to $(\mathbf{I} - \gamma \mathbf{A})^{-1}$ in response to rank-one update $\mathbf{A}_{\star,i} \mathbf{y}^T$ in terms of old $\mathbf{P}$?*

To address this issue, we show a commutative law of $\mathbf{P}$.

Lemma 2. *For any transition matrix $\mathbf{A}$ and its corresponding proximity matrix $\mathbf{P}$, the following property holds:*

$$\mathbf{PA} = \mathbf{AP}.$$

In general, multiplication of matrices is not commutative. However, the commutative property of RWR in Lemma 2 allows us to efficiently compute the changes to $(\mathbf{I} - \gamma \mathbf{A})^{-1}$. Precisely, the changes to $(\mathbf{I} - \gamma \mathbf{A})^{-1}$, as Theorem 2 will show, involve the computation of $\mathbf{PA}$ that is a *matrix-matrix* multiplication, entailing $O(|V|^3)$ time if carried out naively. In contrast, by Lemma 2, computing $\mathbf{PA}$ can be reduced to $O(|V|^2)$ time, requiring only *matrix scaling and subtraction*. This is because Lemma 2 enables $\mathbf{PA}$ to be computed as

$$\mathbf{PA} = \mathbf{AP} = \tfrac{1}{\gamma} \left(\mathbf{P} - (1-\gamma)\mathbf{I}\right) \tag{6}$$

where the last equality holds by rearranging terms in Eq.(1).

Leveraging Lemmas 1 and 2, we can characterize new $\tilde{\mathbf{P}}$.

Theorem 2. *Given old graph $G = (V, E)$ and its old proximity matrix $\mathbf{P}$, after edge (i, j) with $i \in V$ and $j \notin V$ is inserted, the new proximity matrix $\tilde{\mathbf{P}}$ can be updated as*

$$\tilde{\mathbf{P}} = \left[\begin{array}{c|c} \overbrace{\mathbf{P}}^{|V| \textit{ cols}} & \overbrace{\mathbf{0}}^{\textit{col } j} \\ \hline \gamma \mathbf{P}_{i,\star} & 1-\gamma \end{array}\right] \begin{array}{l} \} \ |V| \textit{ rows} \\ \leftarrow \textit{row } j \end{array} \qquad \textit{if } \mathcal{O}(i) = 0; \tag{7}$$

$$\tilde{\mathbf{P}} = \left[\begin{array}{c|c} \overbrace{\mathbf{P} + \frac{\mathbf{z} \cdot \mathbf{P}_{i,\star}}{1-\mathbf{z}_i}}^{|V| \textit{ cols}} & \overbrace{\mathbf{0}}^{\textit{col } j} \\ \hline \frac{\gamma}{\mathcal{O}(i)+1} \left(\frac{\mathbf{P}_{i,\star}}{1-\mathbf{z}_i}\right) & 1-\gamma \end{array}\right] \begin{array}{l} \} \ |V| \textit{ rows} \\ \leftarrow \textit{row } j \end{array} \qquad \textit{if } \mathcal{O}(i) \neq 0. \tag{8}$$

where auxiliary vector $\mathbf{z} = \frac{1}{\mathcal{O}(i)+1} \left(\mathbf{e}_i - \frac{1}{1-\gamma} \mathbf{P}_{\star,i}\right) \in \mathbb{R}^{|V| \times 1}$, $\mathbf{z}_i$ is the i-th entry of $\mathbf{z}$, and $\mathbf{P}_{i,\star}$ is the i-th row of $\mathbf{P}$.

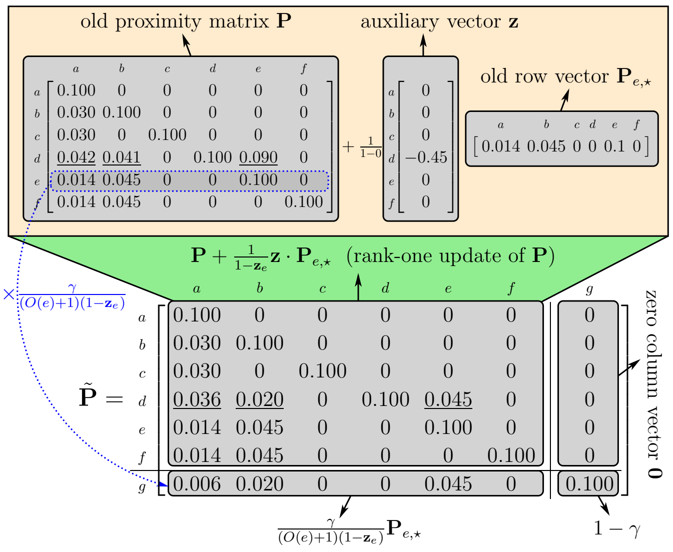
Theorem 2 gives an efficient way to incrementally compute new proximity matrix $\tilde{\mathbf{P}}$ when edge $(i, j)_{i \in V, j \notin V}$ is inserted. When $\mathcal{O}(i) = 0$, it requires $O(|V|)$ time to compute Eq.(7). When $\mathcal{O}(i) \neq 0$, it requires $O(|V|^2)$ time to compute Eq.(8), including: (a) $O(|V|)$ time for vector $\mathbf{z}$; (b) $O(|V|^2)$ time for $(\mathbf{z} \cdot \mathbf{P}_{i,\star})$; (c) $O(|V|)$ time to scale row vector $\mathbf{P}_{i,\star}$.

Example 4. *Recall old graph G in Figure 1 (left) and its old proximity matrix $\mathbf{P}$ (see Example 3). Given $\gamma = 0.9$, after edge (e, g) is added to G, new $\tilde{\mathbf{P}}$ is updated as follows:*

Since $\mathcal{O}(e) = 1 > 0$, we first compute $\mathbf{z}$ by Theorem 2:

$$\mathbf{z} = \tfrac{1}{1+1}\big(\mathbf{e}_e - \tfrac{1}{1-0.9}\mathbf{P}_{\star,e}\big) = \begin{array}{c} \begin{matrix} a & b & c & d & e & f \end{matrix} \\ \left[\begin{matrix} 0 & 0 & 0 & -0.45 & 0 & 0 \end{matrix}\right]^T \end{array}.$$

Then, noting $\mathbf{z}_e = 0$, we can obtain new $\tilde{\mathbf{P}}$ from Eq.(8):



It is worth noting that the $O(|V|^2)$ time of computing $\tilde{\mathbf{P}}$ by Eq.(8) is the *worst-case* complexity, dominated by $(\mathbf{z} \cdot \mathbf{P}_{i,\star})$. Generally, such $O(|V|^2)$ time can be reduced to $O(|\mathbf{z}||\mathbf{P}_{i,\star}|)$[1] by updating only a nonzero subset of $V \times V$ elements of $\mathbf{P}$:

$$\{x \in V : [\mathbf{z}]_x \neq 0\} \times \{y \in V : [\mathbf{P}_{i,\star}]_y \neq 0\} \subseteq V \times V.$$

For instance, to obtain the upper-left block of $\tilde{\mathbf{P}}$ in Example 4, we actually need update only $|\mathbf{z}| \times |\mathbf{P}_{e,\star}| = 1 \times 3 = 3$ entries (underlined) instead of all $|V|^2 = 36$ entries in $\mathbf{P}$.

C. Inserting Edge (i, j) with $i \in V$ and $j \in V$

We next investigate case (C3): the insertion of edge (i, j) with $i \in V$ and $j \in V$. As new $\tilde{\mathbf{A}}$ and old $\mathbf{A}$ are of the same size, it makes sense to denote their change as $\mathbf{\Delta A} := \tilde{\mathbf{A}} - \mathbf{A}$.[2]

To characterize $\mathbf{\Delta A}$, we have the following lemma.

Lemma 3. *Given old graph $G = (V, E)$ and its old transition matrix $\mathbf{A}$, after edge (i, j) with $i \in V$ and $j \in V$ is inserted, the changes $\mathbf{\Delta A}$ can be expressed as*

$$\mathbf{\Delta A} = \mathbf{u}\mathbf{e}_i^T \text{ with } \mathbf{u} := \begin{cases} \mathbf{e}_j & \text{if } \mathcal{O}(i) = 0; \\ \frac{1}{\mathcal{O}(i)+1}(\mathbf{e}_j - \mathbf{A}_{\star,i}) & \text{if } \mathcal{O}(i) \neq 0. \end{cases} \tag{9}$$

Lemma 3 implies that all the nonzeros of $\mathbf{\Delta A}$ appear only in the i-th column of $\mathbf{\Delta A}$ that can be represented as the scaling of old $\mathbf{A}_{\star,i}$ except the j-th entry of $\mathbf{A}_{\star,i}$.

Example 5. *When edge (a, e) is added to G in Figure 1, it follows from $\mathcal{O}(a) = 3$ and $\mathbf{A}_{\star,a} = [0\ \frac{1}{3}\ \frac{1}{3}\ \frac{1}{3}\ 0\ 0]^T$ that*

$$\mathbf{\Delta A} = \mathbf{u}\mathbf{e}_a^T \text{ with } \mathbf{u} = \tfrac{1}{3+1}(\mathbf{e}_e - \mathbf{A}_{\star,a}) = \begin{array}{c} \begin{matrix} a & b & c & d & e & f \end{matrix} \\ \left[\begin{matrix} 0 & -\frac{1}{12} & -\frac{1}{12} & -\frac{1}{12} & \frac{1}{4} & 0 \end{matrix}\right]^T \end{array}.$$

[1] $|\mathbf{x}|$ is the number of nonzeros in $\mathbf{x}$; $[\mathbf{x}]_y$ is y-th entry of $\mathbf{x}$.

[2] Note that in cases (C1), (C2), (C4), $\tilde{\mathbf{A}} - \mathbf{A}$ makes no sense.

The rank-one factorization of $\mathbf{\Delta A}$ in Lemma 3 is exploited to characterize the corresponding proximity changes $\mathbf{\Delta P}$.

Lemma 4. *When edge $(i, j)_{i \in V, j \in V}$ is added to $G = (V, E)$, proximity changes $\mathbf{\Delta P}$ (= new $\tilde{\mathbf{P}}$ − old $\mathbf{P}$) are expressible as*

$$\mathbf{\Delta P} = \mathbf{P}\mathbf{u}\mathbf{v}^T \quad \text{with} \quad \mathbf{v}^T = \big(\tfrac{\gamma}{1-\gamma-\gamma\mathbf{P}_{i,\star}\mathbf{u}}\big)\mathbf{P}_{i,\star} \tag{10}$$

where vector $\mathbf{u}$ is defined by Lemma 3.

Lemma 4 suggests that, for case (C3), $\mathbf{\Delta P}$ is a rank-one matrix, *i.e.*, the product of vector $(\mathbf{P}\mathbf{u})$ and row vector $\mathbf{v}^T$, where $\mathbf{u}$ can be obtained by Eq.(9), and $\mathbf{v}^T$ by scaling $\mathbf{P}_{i,\star}$. Thus, it requires $O(|V|^2)$ total time to compute $\mathbf{\Delta P}$, including (a) $O(|V|)$ time for $\mathbf{u}$ and $\mathbf{v}^T$; (b) $O(|V|^2)$ time for $(\mathbf{P}\mathbf{u})$; and (c) $O(|V|^2)$ time for the product of $(\mathbf{P}\mathbf{u})$ and $\mathbf{v}^T$.

To speed up the computation of $\mathbf{\Delta P}$ in Lemma 4 further, there are two methods: (a) Once $(\mathbf{P}\mathbf{u})$ is computed, $(\mathbf{P}_{i,\star}\mathbf{u})$ in Eq.(10) can be obtained directly from the i-th row of the resulting $(\mathbf{P}\mathbf{u})$. (b) The $O(|V|^2)$ time to compute $(\mathbf{P}\mathbf{u})$ can be significantly reduced to $O(|V|)$ since we observe that $(\mathbf{P}\mathbf{u})$ can be described as a linear combination of only two old "pivot proximity vectors" $\mathbf{P}_{\star,i}$ and $\mathbf{P}_{\star,j}$:

$$\mathbf{P}\mathbf{u} = \square \cdot \mathbf{P}_{\star,i} + \diamondsuit \cdot \mathbf{P}_{\star,j}.$$

To determine scalars $\square$ and $\diamondsuit$, we use the method below.

Theorem 3. *Given old graph $G = (V, E)$, after edge (i, j) with $i \in V$ and $j \in V$ is inserted, proximity changes $\mathbf{\Delta P}$ can be computed as a rank-one matrix:*

$$\mathbf{\Delta P} = \big(\tfrac{1}{1-\gamma-\mathbf{y}_i}\big)\mathbf{y}\mathbf{P}_{i,\star} \quad \text{with} \tag{11}$$

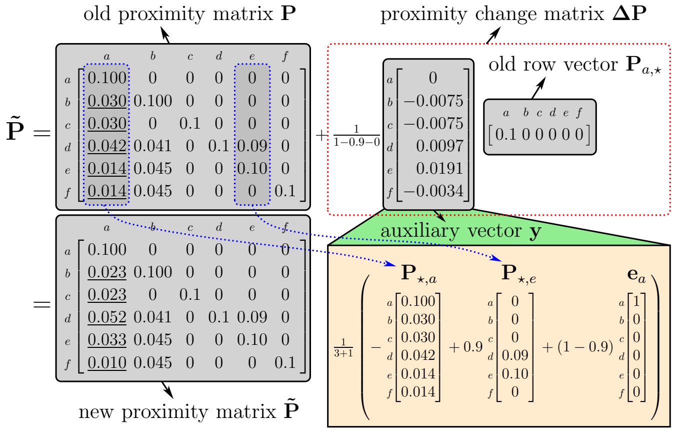
$$\mathbf{y} = \begin{cases} \gamma\mathbf{P}_{\star,j} & \text{if } \mathcal{O}(i) = 0; \\ \frac{1}{\mathcal{O}(i)+1}(\gamma\mathbf{P}_{\star,j} - \mathbf{P}_{\star,i} + (1-\gamma)\mathbf{e}_i) & \text{if } \mathcal{O}(i) \neq 0. \end{cases}$$

Theorem 3 is an optimized version of Lemma 4. It reduces the computation of $(\mathbf{P}\mathbf{u})$ in Lemma 4 from $O(|V|^2)$ to $O(|V|)$. Furthermore, the rank-one structure of $\mathbf{\Delta P}$ in Eq.(11) can reduce the computation of $\mathbf{\Delta P}$ to $O(|\mathbf{y}||\mathbf{P}_{i,\star}|)$ time further, by evaluating only a nonzero subset of $V \times V$ entries of $\mathbf{\Delta P}$:

$$\{x \in V : [\mathbf{y}]_x \neq 0\} \times \{y \in V : [\mathbf{P}_{i,\star}]_y \neq 0\} \subseteq V \times V.$$

Example 6. *Recall old graph G in Figure 1 (left) and its old proximity matrix $\mathbf{P}$ (see below). Given $\gamma = 0.9$, after edge (a, e) is inserted to G, new $\tilde{\mathbf{P}}$ can be updated as follows:*

As $\mathcal{O}(a) = 3$, we first obtain $\mathbf{y}$ and then $\mathbf{\Delta P}$ by Eq.(11):



Algorithm 1: Unit Insertion

```
Input  : old graph $G=(V,E)$, edge $(i,j)$ to be inserted,
         old proximity matrix $\mathbf{P}$ in $G$, and decay factor $\gamma$.
Output: new proximity matrix $\tilde{\mathbf{P}}$ in $G \cup \{(i,j)\}$.
1  if $i \notin V$ and $j \in V$ then                              // Case (C1)
2      update $\tilde{\mathbf{P}} := \begin{array}{c|c|l} |V|\text{ cols} & \text{col } i & \\ \hline \mathbf{P} & \gamma\mathbf{P}_{\star,j} & \}\ |V|\text{ rows} \\ \hline \mathbf{0} & 1-\gamma & \leftarrow \text{row } i \end{array}$
3  else if $i \in V$ and $j \notin V$ then                         // Case (C2)
4      if $\mathcal{O}(i) \neq 0$ then
5          set $\mathbf{z} := \frac{1}{\mathcal{O}(i)+1}(\mathbf{e}_i - \frac{1}{1-\gamma}\mathbf{P}_{\star,i})$
6          update $\tilde{\mathbf{P}} := \begin{array}{c|c|l} |V|\text{ cols} & \text{col } j & \\ \hline \mathbf{P} + \frac{\mathbf{z}\cdot\mathbf{P}_{i,\star}}{1-\mathbf{z}_i} & \mathbf{0} & \}\ |V|\text{ rows} \\ \hline \frac{\gamma}{\mathcal{O}(i)+1}\left(\frac{\mathbf{P}_{i,\star}}{1-\mathbf{z}_i}\right) & 1-\gamma & \leftarrow \text{row } j \end{array}$
7      else
8          update $\tilde{\mathbf{P}} := \begin{array}{c|c|l} |V|\text{ cols} & \text{col } j & \\ \hline \mathbf{P} & \mathbf{0} & \}\ |V|\text{ rows} \\ \hline \gamma\mathbf{P}_{i,\star} & 1-\gamma & \leftarrow \text{row } j \end{array}$
9  else if $i \in V$ and $j \in V$ then                            // Case (C3)
10     if $\mathcal{O}(i) = 0$ then set $\mathbf{y} := \gamma\mathbf{P}_{\star,j}$
11     else set $\mathbf{y} := \frac{1}{\mathcal{O}(i)+1}(\gamma\mathbf{P}_{\star,j} - \mathbf{P}_{\star,i} + (1-\gamma)\mathbf{e}_i)$
12     update $\tilde{\mathbf{P}} := \mathbf{P} + \left(\frac{1}{1-\gamma-\mathbf{y}_i}\right)\mathbf{y}\mathbf{P}_{i,\star}$
13 else if $i \notin V$ and $j \notin V$ then                      // Case (C4)
14     update $\tilde{\mathbf{P}} := \begin{array}{c|cc|l} |V|\text{ cols} & \text{col } i & \text{col } j & \\ \hline \mathbf{P} & \mathbf{0} & \mathbf{0} & \}\ |V|\text{ rows} \\ \hline \mathbf{0} & 1-\gamma & 0 & \leftarrow \text{row } i \\ \mathbf{0} & (1-\gamma)\gamma & 1-\gamma & \leftarrow \text{row } j \end{array}$
```

It is worth noticing that, to efficiently compute $\tilde{\mathbf{P}}$, we need update only $|\mathbf{y}| \times |\mathbf{P}_{a,\star}| = 5 \times 1 = 5$ entries (underlined) instead of all $|V|^2 = 6^2 = 36$ entries in $\mathbf{P}$. □

D. Inserting Edge (i,j) with $i \notin V$ and $j \notin V$

We next handle case (C4): inserting edge (i,j) with $i \notin V$ and $j \notin V$. After insertion, new transition matrix $\tilde{\mathbf{A}}$ is

$$\tilde{\mathbf{A}} = \begin{array}{c|cc|l} |V|\text{ cols} & \text{col } i & \text{col } j & \\ \hline \mathbf{A} & \mathbf{0} & \mathbf{0} & \}\ |V|\text{ rows} \\ \hline \mathbf{0} & 0 & 0 & \leftarrow \text{row } i \\ \mathbf{0} & 1 & 0 & \leftarrow \text{row } j \end{array} \tag{12}$$

Based on the block diagonal structure of $\tilde{\mathbf{A}}$, new $\tilde{\mathbf{P}}$ can be expressed in a block diagonal form as well, as shown below.

Theorem 4. *Given graph $G=(V,E)$ and its old proximity matrix $\mathbf{P}$, after edge (i,j) with $i \notin V$ and $j \notin V$ is inserted, new proximity matrix $\tilde{\mathbf{P}}$ can be computed as*

$$\tilde{\mathbf{P}} = \begin{array}{c|cc|l} |V|\textit{ cols} & \textit{col } i & \textit{col } j & \\ \hline \mathbf{P} & \mathbf{0} & \mathbf{0} & \}\ |V|\textit{ rows} \\ \hline \mathbf{0} & 1-\gamma & 0 & \leftarrow \textit{row } i \\ \mathbf{0} & (1-\gamma)\gamma & 1-\gamma & \leftarrow \textit{row } j \end{array}$$

Theorem 4 implies that, the insertion of edge $(i,j)_{i\notin V, j\notin V}$ for case (C4) will form another new component in the graph. After edge insertion, the upper-left block of new $\tilde{\mathbf{P}}$ remains unchanged as there are no edges across the two components. Likewise, the upper-right and lower-left blocks of $\tilde{\mathbf{P}}$ are 0s.

Algorithm 2: Unit Deletion

```
Input  : old graph $G=(V,E)$, edge $(i,j)$ to be deleted,
         old proximity matrix $\mathbf{P}$ in $G$, and decay factor $\gamma$.
Output: new proximity matrix $\tilde{\mathbf{P}}$ in $G - \{(i,j)\}$.
1  if $\mathcal{O}(i) = 1$ then  set $\mathbf{y} := -\gamma\mathbf{P}_{\star,j}$
2  else  set $\mathbf{y} := \frac{1}{\mathcal{O}(i)-1}(\mathbf{P}_{\star,i} - \gamma\mathbf{P}_{\star,j} - (1-\gamma)\mathbf{e}_i)$
3  update $\tilde{\mathbf{P}} := \mathbf{P} + \frac{1}{(1-\gamma-\mathbf{y}_i)}\mathbf{y}\mathbf{P}_{i,\star}$
4  if $i$ or $j$ is an isolated node after deletion then delete $i$ or $j$
```

E. Incremental Algorithm for Unit Insertion

To summarize the cases (C1)–(C4) in Sections III-A–III-D, Algorithm 1 gives a complete scheme which can incrementally compute all pairs of RWR proximities for unit insertion.

The correctness of Algorithm 1 is shown by Theorems 1–4, corresponding to 4 cases: (C1) (Lines 1–2), (C2) (Lines 3–8), (C3) (Lines 9–12), and (C4) (Lines 13–14), respectively.

For computational cost, we have the following result.

Theorem 5. *For any edge inserted to graph $G=(V,E)$, it requires $O(|V|^2)$ worst-case time and $O(|V|^2)$ memory to incrementally compute all pairs of proximities accurately.*

The $O(|V|^2)$ worst-case time, in general, can be reduced to $O(\max\{|V|, |\mathbf{z}||\mathbf{P}_{i,\star}|, |\mathbf{y}||\mathbf{P}_{i,\star}|\})$ time if we skip all 0 entries of $\mathbf{z}, \mathbf{y}, \mathbf{P}_{i,\star}$ to compute $\mathbf{z}\mathbf{P}_{i,\star}$ and $\mathbf{y}\mathbf{P}_{i,\star}$ (see Example 6).

F. Decremental Algorithm for Unit Deletion

Unlike edge insertion that is divided into cases (C1)–(C4), we focus only on one case for edge deletion: Given old graph $G=(V,E)$, the removal of edge (i,j) with $i \in V$ and $j \in V$, since we can first assume that the deletion of edge (i,j) would not remove its end nodes i and j. If i or j becomes an isolated node (whose in- and out-degrees are all 0s) after edge deletion, then we can remove i or j later.

Algorithm 2 provides a decremental way to update all-pairs proximities for unit deletion. The proofs of its correctness and complexity are similar to those of Theorem 3.

IV. Avoid Memoizing All-Pairs RWR

In the previous section, the $O(|V|^2)$ memory is dominated by storing all pairs of new/old proximities. To avoid $O(|V|^2)$ memory, we next propose our partitioning techniques that can update each segment of $\mathbf{P}$ independently.

Due to space limitations, we focus only on our partitioning methods to update $\mathbf{P}$ in case (C2) for $\mathcal{O}(i) \neq 0$, *i.e.*, Eq.(8), as this is the most complicated case among (C1)–(C4).

Our central idea of avoiding $O(|V|^2)$ memory is to partition $\mathbf{P} \in \mathbb{R}^{|V|\times|V|}$ and $\mathbf{z} \in \mathbb{R}^{|V|\times 1}$ into $\lceil \frac{|V|}{l} \rceil$ segments of size $l \times |V|$ and $l \times 1$, respectively (except for the last segment that may be smaller), where $1 \leq l \leq |V|$ is a user-controlled integer that makes each segment small enough to fit in memory. After partitioning, $\mathbf{P}$ and $\mathbf{z}$ becomes

$$\mathbf{P} = \begin{array}{c|l} |V|\text{ cols} & \\ \hline [\mathbf{P}]_1 & \}\ l\text{ rows} \\ \hline [\mathbf{P}]_2 & \}\ l\text{ rows} \\ \hline \vdots & \vdots \\ \hline [\mathbf{P}]_N & \}\ (|V|-(N-1)l)\text{ rows} \end{array} \qquad \mathbf{z} = \begin{array}{c|l} |V|\text{ cols} & \\ \hline [\mathbf{z}]_1 & \}\ l\text{ rows} \\ \hline [\mathbf{z}]_2 & \}\ l\text{ rows} \\ \hline \vdots & \vdots \\ \hline [\mathbf{z}]_N & \}\ (|V|-(N-1)l)\text{ rows} \end{array} \quad \text{with } N = \left\lceil \frac{|V|}{l} \right\rceil$$

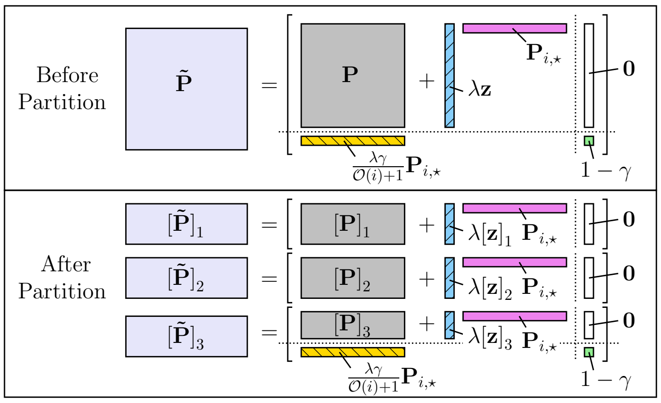


Figure 2: Compute $\tilde{\mathbf{P}}$ in Eq.(8) Segment-wisely by Eqs.(13)–(14)

where $[\mathbf{P}]_x$ is the x-th segment ($l \times |V|$) of $\mathbf{P}$ ($1 \le x \le N-1$), and $[\mathbf{z}]_x$ is the x-th segment ($l \times 1$) of $\mathbf{z}$.

As the upper-left block ($\mathbf{P} + \frac{\mathbf{z} \cdot \mathbf{P}_{i,\star}}{1-\mathbf{z}_i}$) of new $\tilde{\mathbf{P}}$ is a rank-one update of old $\mathbf{P}$, it can be computed segment-wisely as

$$\mathbf{P} + \lambda \mathbf{z} \cdot \mathbf{P}_{i,\star} = \begin{array}{c} |V| \text{ cols} \\ \left[\begin{array}{c} \overbrace{[\mathbf{P}]_1 + \lambda [\mathbf{z}]_1 \cdot \mathbf{P}_{i,\star}} \\ \hline [\mathbf{P}]_2 + \lambda [\mathbf{z}]_2 \cdot \mathbf{P}_{i,\star} \\ \hline \vdots \\ \hline [\mathbf{P}]_N + \lambda [\mathbf{z}]_N \cdot \mathbf{P}_{i,\star} \end{array}\right] \end{array} \begin{array}{l} \\ \} \; l \text{ rows} \\ \} \; l \text{ rows} \\ \vdots \\ \} \; (|V| - (N-1)l) \text{ rows} \end{array} \quad \text{with } \lambda = \frac{1}{1-\mathbf{z}_i}.$$

This suggests that, to incrementally evaluate new $\tilde{\mathbf{P}}$, we just need load $\mathbf{P}_{i,\star}$ and one segment of $\mathbf{P}$, say $[\mathbf{P}]_x$, into memory at one time; and each new segment $[\tilde{\mathbf{P}}]_x$ can be updated from old segment $[\mathbf{P}]_x$ independently as follows:

$$[\tilde{\mathbf{P}}]_x = \Big[\overbrace{[\mathbf{P}]_x + \lambda [\mathbf{z}]_x \cdot \mathbf{P}_{i,\star}}^{|V| \text{ cols}} \Big| \overbrace{\mathbf{0}}^{\text{col } j} \Big] \Big\} \; l \text{ rows} \qquad (\forall x = 1, \cdots, N)$$
$$\text{with } \lambda = \frac{1}{1 - \frac{1}{\mathcal{O}(i)+1}(1 - \frac{1}{1-\gamma}\mathbf{P}_{i,i})} \text{ and } [\mathbf{z}]_x = \frac{1}{\mathcal{O}(i)+1}([\mathbf{e}_i]_x - \frac{1}{1-\gamma}[\mathbf{P}_{\star,i}]_x). \tag{13}$$

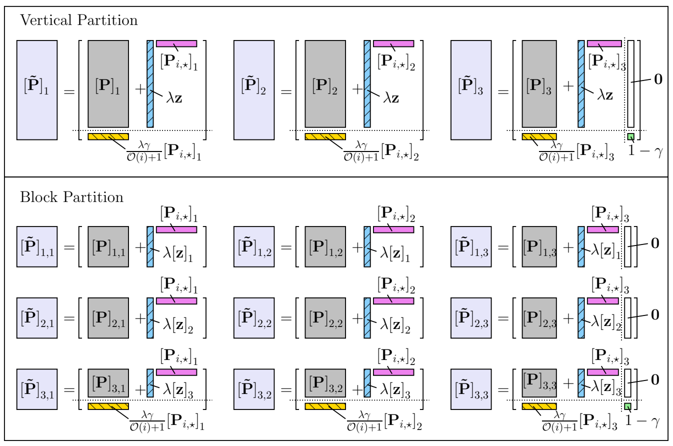
except for the last segment being

$$[\tilde{\mathbf{P}}]_N = \left[\begin{array}{c|c} \overbrace{[\mathbf{P}]_N + \lambda [\mathbf{z}]_N \cdot \mathbf{P}_{i,\star}}^{|V| \text{ cols}} & \overbrace{\mathbf{0}}^{\text{col } j} \\ \hline \frac{\lambda\gamma}{\mathcal{O}(i)+1}\mathbf{P}_{i,\star} & 1-\gamma \end{array}\right] \begin{array}{l} \} \; (|V| - (N-1)l - 1) \text{ rows} \\ \leftarrow \text{row } j \end{array} \tag{14}$$

The advantage of our partitioning method Eqs.(13)–(14) is that it requires only $O(l|V|)$ memory and $O(\lceil \frac{|V|}{l} \rceil)$ I/O costs to incrementally update $\mathbf{P}$, with no need of $O(|V|^2)$ memory to load the entire $\mathbf{P}$. Moreover, each segment of $\mathbf{P}$ is updated independently. Figure 2 depicts how our partitioning method Eqs.(13)–(14) segment-wisely updates $\mathbf{P}$.

The integer $1 \le l \le |V|$ is a user-controlled parameter that is a trade-off balancing memory and I/O costs. For instance, when $l = 1$, $\mathbf{P}$ can be row-by-row loaded and updated in just $O(|V|)$ memory, but requires $O(|V|)$ I/O costs in total for $|V|$ rows update; when $l = |V|$, it requires only $O(1)$ I/O cost in total for all pairs of inputs/outputs, but entails $O(|V|^2)$ memory to load the entire $\mathbf{P}$ — this reduces to the *in-memory* algorithms that we discussed in Section III.

The CPU time for updating each segment of $\mathbf{P}$ in Eqs.(13)–(14) is $O(l|V|)$ in the worst case, which in practice can be reduced to $O(|[\mathbf{z}]_x||\mathbf{P}_{i,\star}|)$ further if zero entries in vectors $[\mathbf{z}]_x$ and $\mathbf{P}_{i,\star}$ are skipped. In total, since there are $\lceil \frac{|V|}{l} \rceil$ segments, the CPU time to update all ($|V|^2$) pairs of $\mathbf{P}$ retains $O(|V|^2)$



Vertical Partition
Block Partition
$[\tilde{\mathbf{P}}]_1$
$[\tilde{\mathbf{P}}]_2$
$[\tilde{\mathbf{P}}]_3$
$[\mathbf{P}]_1$
$[\mathbf{P}]_2$
$[\mathbf{P}]_3$
$[\mathbf{P}_{i,\star}]_1$
$[\mathbf{P}_{i,\star}]_2$
$[\mathbf{P}_{i,\star}]_3$
$\lambda\mathbf{z}$
$\mathbf{0}$
$\frac{\lambda\gamma}{\mathcal{O}(i)+1}[\mathbf{P}_{i,\star}]_1$
$\frac{\lambda\gamma}{\mathcal{O}(i)+1}[\mathbf{P}_{i,\star}]_2$
$\frac{\lambda\gamma}{\mathcal{O}(i)+1}[\mathbf{P}_{i,\star}]_3$
$1-\gamma$
$[\tilde{\mathbf{P}}]_{1,1}$
$[\tilde{\mathbf{P}}]_{1,2}$
$[\tilde{\mathbf{P}}]_{1,3}$
$[\tilde{\mathbf{P}}]_{2,1}$
$[\tilde{\mathbf{P}}]_{2,2}$
$[\tilde{\mathbf{P}}]_{2,3}$
$[\tilde{\mathbf{P}}]_{3,1}$
$[\tilde{\mathbf{P}}]_{3,2}$
$[\tilde{\mathbf{P}}]_{3,3}$
$[\mathbf{P}]_{1,1}$
$[\mathbf{P}]_{1,2}$
$[\mathbf{P}]_{1,3}$
$[\mathbf{P}]_{2,1}$
$[\mathbf{P}]_{2,2}$
$[\mathbf{P}]_{2,3}$
$[\mathbf{P}]_{3,1}$
$[\mathbf{P}]_{3,2}$
$[\mathbf{P}]_{3,3}$
$\lambda[\mathbf{z}]_1$
$\lambda[\mathbf{z}]_2$
$\lambda[\mathbf{z}]_3$

Figure 3: Compute $\tilde{\mathbf{P}}$ in Eq.(8) via Vertical Partitioning into 3 Segments or via Block Partitioning into 3×3 Segments

in the worst case, and $O(\sum_{x=1}^{N} |[\mathbf{z}]_x||\mathbf{P}_{i,\star}|) = O(|\mathbf{z}||\mathbf{P}_{i,\star}|)$ in practice, which is the same as Algorithm 1.

In addition to the horizontal partitioning in Eqs.(13)–(14), we can similarly devise vertical partitioning and block partitioning techniques to incrementally evaluate $\mathbf{P}$, as picturized in Figure 3. Their performance will be compared in Section VI.

V. Bulk Updates

We consider two types of bulk updates: a) *pure bulk updates*: only one type of updates, insertions or deletions, is allowed; b) *mixed bulk updates*: a mixture of insertions & deletions.

Pure Bulk Insertions. Given a set of edges to be inserted to old $G = (V, E)$, *i.e.*, $\Delta G := \{(i_1, j_1), (i_2, j_2), \cdots, (i_\delta, j_\delta)\}$, where i_k and j_k ($1 \le k \le \delta$) can be new/old nodes in V, the traditional method to compute new $\tilde{\mathbf{P}}$ in $G \cup \Delta G$ requires repeated execution of unit insertion (Algorithm 1) for δ times, and may produce many unnecessary intermediate updates.

However, we observe that, for pure bulk updates, the order of edge insertions in ΔG is irrelevant to new $\tilde{\mathbf{P}}$ in $G \cup \Delta G$; and, in general, there are often many repeated nodes in ΔG. This gives us a chance to handle multiple edges in bulk.

Our main idea is to sort all edges $\{(i_k, j_k)\}$ in ΔG by its head node i_k into several groups $\{\Delta G_{i_k}\}$. Then, all edges in each group ΔG_{i_k} are divided into at most 2 subgroups: $\Delta G_{i_k}^1$ and $\Delta G_{i_k}^2$, according to whether its tail node $j_k \in V$.

Example 7. $\Delta G = \{(a, e), (e, g), (e, l), (e, f), (h, b), (e, m)\}$ *in Figure 1 can be divided into three groups:* $\Delta G_a = \{(a, e)\}$, $\Delta G_e = \{(e, g), (e, f), (e, m), (e, l)\}$ *and* $\Delta G_h = \{(h, b)\}$, *where* ΔG_e *can be partitioned into two subgroups further:* $\Delta G_e^1 = \{(e, f)\}$ *and* $\Delta G_e^2 = \{(e, g), (e, m), (e, l)\}$. □

The main advantage of dividing ΔG is that, after division, all the insertions in each group can be handled *simultaneously*. To elaborate on this, let us focus on one group ΔG_i:

$$\Delta G_i := \{(i, x)\}_{\forall x \in J} \quad \text{with} \quad J := \{j_1, \cdots, j_\delta\}.$$

Analogous to unit insertion in Section III, for every group, we classify new insertions ΔG_i to $G = (V, E)$ into 4 cases:

(C1) $i \notin V$, $j_1 \in V, \cdots, j_\delta \in V$; (C2) $i \in V$, $j_1 \notin V, \cdots, j_\delta \notin V$;

Algorithm 3: Pure Bulk Insertions

```
Input  : old graph $G = (V, E)$, decay factor $\gamma$,
         a set of edges $\Delta G := \{(i_k, j_k)\}$ to be inserted,
         old proximity matrix $\mathbf{P}$ in $G$.
Output: new proximity matrix $\tilde{\mathbf{P}}$ in $G \cup \Delta G$.
repeat
1   sort all edges $\{(i_k, j_k)\}$ of $\Delta G$ into $|I|$ groups $\{\Delta G_i\}$
      first by $i_k$ and then by whether $j_k$ is an old node in $G$.
2   set $\Delta G_{i_{\max}}$ := one of the groups with the maximum
      number of edges in $\{\Delta G_i\}$.
3   set $J := \{$node $j : (i, j) \in \Delta G_{i_{\max}}\}$ and $\delta := |J|$.
4   update new $\tilde{\mathbf{P}}$ in $G \cup \Delta G_{i_{\max}}$ from old $\mathbf{P}$ in $G$, according
      to the last column of Table II.
5   update $\Delta G := \Delta G - \Delta G_{i_{\max}}$ and $G := G \cup \Delta G_{i_{\max}}$
until $\Delta G := \varnothing$
```

Case	New Transition Matrix $\tilde{\mathbf{A}}$	New Proximity Matrix $\tilde{\mathbf{P}}$
(C1): $i \notin V$ $j_1 \in V$ $\cdots$ $j_\delta \in V$	$\begin{array}{ccl} \overbrace{}^{\lvert V\rvert \text{ cols}} & \overbrace{}^{\text{col } i} & \\ \mathbf{A} & (1/\delta)\mathbf{e}_J & \} \ \lvert V\rvert \text{ rows} \\ \hline \mathbf{0} & 0 & \leftarrow \text{row } i \end{array}$ with $(\mathbf{e}_J)_x := \begin{cases} 1, & x \in J \\ 0, & x \notin J \end{cases}$	$\begin{array}{ccl} \overbrace{}^{\lvert V\rvert \text{ cols}} & \overbrace{}^{\text{col } i} & \\ \mathbf{P} & \frac{\gamma}{\delta}\sum_{j \in J} \mathbf{P}_{\star,j} & \} \ \lvert V\rvert \text{ rows} \\ \hline \mathbf{0} & 1-\gamma & \leftarrow \text{row } i \end{array}$
(C2): $i \in V$ $j_1 \notin V$ $\cdots$ $j_\delta \notin V$	① if $\mathcal{O}(i) = 0$, then $\tilde{\mathbf{A}} :=$ $\begin{array}{ccl} \overbrace{}^{\lvert V\rvert \text{ cols}} & \overbrace{}^{\delta \text{ cols}} & \\ \mathbf{A} & \mathbf{0} & \} \ \lvert V\rvert \text{ rows} \\ \hline (1/\delta)\mathbf{1}_\delta \mathbf{e}_i^T & \mathbf{0} & \} \ \delta \text{ rows} \end{array}$ with $\mathbf{1}_\delta := \underbrace{[1, 1, \cdots, 1]}_{\delta}^T$ ② if $\mathcal{O}(i) \neq 0$, then $\tilde{\mathbf{A}} :=$ $\begin{array}{ccl} \overbrace{}^{\lvert V\rvert \text{ cols}} & \overbrace{}^{\delta \text{ cols}} & \\ \mathbf{A} + \mathbf{v}\mathbf{e}_i^T & \mathbf{0} & \} \ \lvert V\rvert \text{ rows} \\ \hline \frac{1}{\mathcal{O}(i)+\delta}\mathbf{1}_\delta \mathbf{e}_i^T & \mathbf{0} & \} \ \delta \text{ rows} \end{array}$ with $\mathbf{v} := -\frac{\delta}{\mathcal{O}(i)+\delta}\mathbf{A}_{\star,i}$	① if $\mathcal{O}(i) = 0$, then $\tilde{\mathbf{P}} :=$ $\begin{array}{ccl} \overbrace{}^{\lvert V\rvert \text{ cols}} & \overbrace{}^{\delta \text{ cols}} & \\ \mathbf{P} & \mathbf{0} & \} \ \lvert V\rvert \text{ rows} \\ \hline \frac{\gamma}{\delta}\mathbf{1}_\delta \mathbf{P}_{i,\star} & (1-\gamma)\mathbf{I} & \} \ \delta \text{ rows} \end{array}$ ② if $\mathcal{O}(i) \neq 0$, then $\tilde{\mathbf{P}} :=$ $\begin{array}{ccl} \overbrace{}^{\lvert V\rvert \text{ cols}} & \overbrace{}^{\delta \text{ cols}} & \\ \mathbf{P} + \frac{\mathbf{z}\mathbf{P}_{i,\star}}{1-\mathbf{z}_i} & \mathbf{0} & \} \ \lvert V\rvert \text{ rows} \\ \hline \frac{\gamma \mathbf{1}_\delta \mathbf{P}_{i,\star}}{(\mathcal{O}(i)+\delta)(1-\mathbf{z}_i)} & (1-\gamma)\mathbf{I} & \} \ \delta \text{ rows} \end{array}$ with $\mathbf{z} := \frac{\delta}{\mathcal{O}(i)+\delta}(\mathbf{e}_i - \frac{1}{1-\gamma}\mathbf{P}_{\star,i})$
(C3): $i \in V$ $j_1 \in V$ $\cdots$ $j_\delta \in V$	$\tilde{\mathbf{A}} := \mathbf{A} + \mathbf{u}\mathbf{e}_i^T$ with ① if $\mathcal{O}(i) = 0$, then $\mathbf{u} := (1/\delta)\mathbf{e}_J$ ② if $\mathcal{O}(i) \neq 0$, then $\mathbf{u} := \frac{\delta}{\mathcal{O}(i)+\delta}(\frac{1}{\delta}\mathbf{e}_J - \mathbf{A}_{\star,i})$	$\tilde{\mathbf{P}} := \mathbf{P} + \frac{1}{1-\gamma-\mathbf{y}_i}\mathbf{y}\mathbf{P}_{i,\star}$ with ① if $\mathcal{O}(i) = 0$, then $\mathbf{y} := \frac{\gamma}{\delta}\sum_{j \in J} \mathbf{P}_{\star,j}$ ② if $\mathcal{O}(i) \neq 0$, then $\mathbf{y} := \frac{1}{\mathcal{O}(i)+\delta}(\gamma \sum_{j \in J} \mathbf{P}_{\star,j} - \delta \mathbf{P}_{\star,i}$ $+\delta(1-\gamma)\mathbf{e}_i)$
(C4): $i \notin V$ $j_1 \notin V$ $\cdots$ $j_\delta \notin V$	$\begin{array}{cccl} \overbrace{}^{\lvert V\rvert \text{ cols}} & \overbrace{}^{\text{col } i} & \overbrace{}^{\delta \text{ cols}} & \\ \mathbf{A} & \mathbf{0} & \mathbf{0} & \} \ \lvert V\rvert \text{ rows} \\ \hline \mathbf{0} & 0 & \mathbf{0} & \leftarrow \text{row } i \\ \mathbf{0} & \frac{1}{\delta}\mathbf{1}_\delta & \mathbf{0} & \} \ \delta \text{ rows} \end{array}$	$\begin{array}{cccl} \overbrace{}^{\lvert V\rvert \text{ cols}} & \overbrace{}^{\text{col } i} & \overbrace{}^{\delta \text{ cols}} & \\ \mathbf{P} & \mathbf{0} & \mathbf{0} & \} \ \lvert V\rvert \text{ rows} \\ \hline \mathbf{0} & 1-\gamma & \mathbf{0} & \leftarrow \text{row } i \\ \mathbf{0} & \frac{(1-\gamma)\gamma}{\delta}\mathbf{1}_\delta & (1-\gamma)\mathbf{I} & \} \ \delta \text{ rows} \end{array}$

Table II: New $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{P}}$ for Four Cases of Bulk Insertions

(C3) $i \in V,\ j_1 \in V, \cdots, j_\delta \in V$; (C4) $i \notin V,\ j_1 \notin V, \cdots, j_\delta \notin V$.

Similar to unit insertion, for each case of bulk insertions, we can obtain new $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{P}}$ in response to ΔG_i, by extending Theorems 1–4 to bulk insertions. As shown in Table II, both new $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{P}}$ in response to $\Delta G_i = \{(i, x)\}_{\forall x \in J}$ also bear rank-one update structures.

Table II implies an incremental algorithm to compute $\tilde{\mathbf{P}}$ for pure bulk insertions, as shown in Algorithm 3. The *worst-case* complexity of Algorithm 3 are analyzed below.

Theorem 6. *Let $|\tilde{V}|$ be the total number of nodes in new graph $G \cup \Delta G$, $|\Delta G|$ be the number of inserted edges in ΔG, and $|I|$ be the total number of groups in ΔG (Lines 1–2). In the worst case, Algorithm 3 entails $O(|I||\tilde{V}|^2 + |\Delta G||\tilde{V}|)$ time and $O(|\tilde{V}|^2)$ memory for ΔG bulk insertions.*

The *actual* running time of Algorithm 3 is even faster than its worst-case time in Theorem 6, due to two reasons: (a) The type of edge insertions, in practice, may not *always* meet the most time-consuming cases (C2) and (C3); (b) For each edge update (i, j), the $O(|\tilde{V}|^2)$ worst-case time in cases (C2) and (C3) is dominated by the vector products $\mathbf{z}\mathbf{P}_{i,\star}$ and $\mathbf{y}\mathbf{P}_{i,\star}$, which, in practice, can be reduced to $O(\max\{|\mathbf{z}|, |\mathbf{y}|\}|\mathbf{P}_{i,\star}|)$ time further, by eliminating 0 entries in vectors $\mathbf{y}$, $\mathbf{z}$, $\mathbf{P}_{i,\star}$.

Algorithm 4: Pure Bulk Deletions

```
Input  : the same as Algorithm 3 except for
         "a set of edges $\Delta G := \{(i_k, j_k)\}$ to be deleted"
Output: new proximity matrix $\tilde{\mathbf{P}}$ in $G - \Delta G$.
1 sort all edges $\{(i_k, j_k)\}$ of $\Delta G$ by $i_k$ into $|I|$ groups: $\{\Delta G_i\}$.
2 foreach group $\Delta G_i$ in $\Delta G$ do
3     set $J := \{$node $j : (i, j) \in \Delta G_i\}$ and $\delta := |J|$.
4     if $\mathcal{O}(i) = 1$ then $\mathbf{y} := -\frac{\gamma}{\delta}\sum_{j \in J} \mathbf{P}_{\star,j}$
5     else $\mathbf{y} := \frac{1}{\mathcal{O}(i)-\delta}(\delta \mathbf{P}_{\star,i} - \gamma \sum_{j \in J} \mathbf{P}_{\star,j} - \delta(1-\gamma)\mathbf{e}_i)$
6     update $\tilde{\mathbf{P}} := \mathbf{P} + \frac{1}{(1-\gamma-\mathbf{y}_i)}\mathbf{y}\mathbf{P}_{i,\star}$.
7     if $i$ or $j_1$ or $\cdots$ or $j_\delta$ is an isolated node then
          delete node $i$ or $j_1$ or $\cdots$ or $j_\delta$.
```

Algorithm 5: Mixed Bulk Updates

```
Input  : the same as Algorithm 3 except for
         "a set of edges $\Delta G := \{(i_k, j_k, \pm)\}$ to be updated"
Output: new proximity matrix $\tilde{\mathbf{P}}$ in $G \oplus \Delta G$.
1 obtain a set of net updates $\Delta G_{\min}$ from $\Delta G$ via hashing
2 divide $\Delta G_{\min}$ by update type into $\Delta G^-_{\min}$ and $\Delta G^+_{\min}$
3 call Pure Bulk Deletions (Alg. 4) to update $\tilde{\mathbf{P}}$ w.r.t. $\Delta G^-_{\min}$
4 call Pure Bulk Insertions (Alg. 3) to update $\tilde{\mathbf{P}}$ w.r.t. $\Delta G^+_{\min}$
```

Pure Bulk Deletions. For bulk deletions, we first sort all edges $\{(i_k, j_k)\}$ in ΔG by its head node i_k into $|I|$ groups $\{\Delta G_i\}$. To get new $\tilde{\mathbf{P}}$, unlike bulk insertions that split edge types into 4 cases, we just need consider one case: the deletion of $\Delta G_i := \{(i, j_1), \cdots, (i, j_\delta)\}$ with $i \in V, j_1 \in V, \cdots, j_\delta \in V$ from old graph $G = (V, E)$ because, if i or j_1 or $\cdots$ or j_k is an isolated node after deletions, we can remove it later. Algorithm 4 shows our method for bulk deletions. Its cost is the same as Algorithm 3 (replace $|\tilde{V}|$ by $|V|$).

Mixed Bulk Insertions & Deletions. We eliminate from ΔG many unnecessary updates that may "cancel" each other. Our main idea is to get a *net* update set $\Delta G_{\min}$ via a hash table to count occurrences of updates in ΔG. Precisely, for each edge update (hash key) in ΔG, we first initialize its count (hash value) with 0, and then increase (*resp.* decrease) its count by 1 when an insertion (*resp.* deletion) is in ΔG. Lastly, all hash keys with nonzero counts in ΔG make $\Delta G_{\min}$ such that $G \oplus \Delta G_{\min} = G \oplus \Delta G$ yet $|\Delta G_{\min}| \ll |\Delta G|$.

Algorithm 5 provides an efficient algorithm for mixed bulk updates. It requires $O(|I_{\max}||\tilde{V}|^2 + |\Delta G_{\min}||\tilde{V}|)$ time, where $|I_{\max}|$ is the maximum number $|I|$ of groups for pure bulk updates $\Delta G^-_{\min}$ and $\Delta G^+_{\min}$, and $|\tilde{V}|$ is number of nodes in $G \cup \Delta G^+_{\min}$. The memory is $O(l|\tilde{V}|)$ with $O(\lceil \frac{|\tilde{V}|}{l} \rceil)$ I/Os.

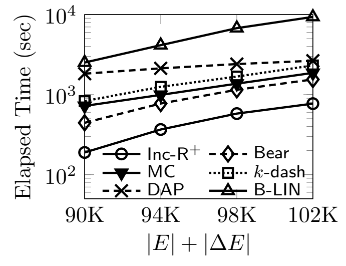
VI. Experiments

A. Experimental Settings

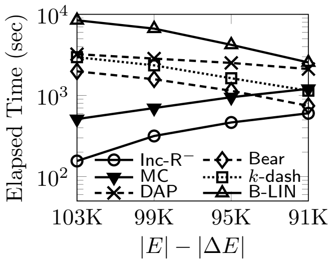
1) Real-life Datasets. We use 4 real datasets, including 2 temporal graphs (DBLP[3], HepPh), and 2 static graphs (Wiki, Email)[4] with synthetic updates simulating real evolutions:

[3]http://dblp.uni-trier.de/xml/

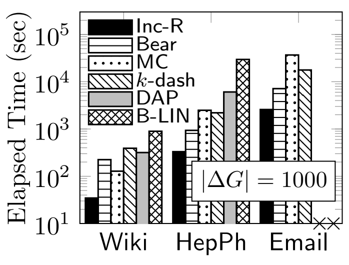
[4]http://snap.stanford.edu/data/index.html



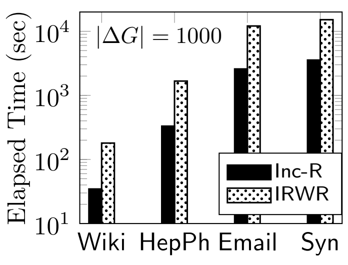
(a) Insertion on DBLP



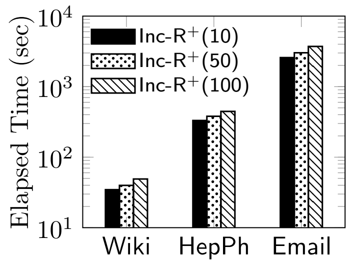
(b) Deletion on DBLP



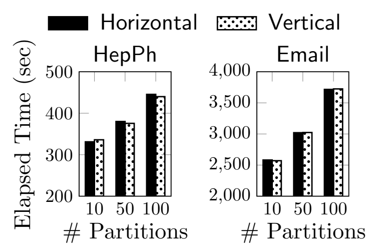
(c) Mixed Update on Real Data



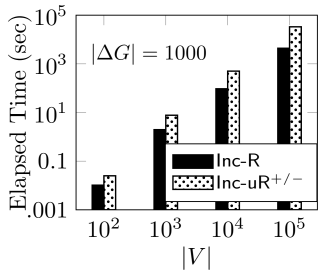
(d) Time for Inc-R *vs.* IRWR



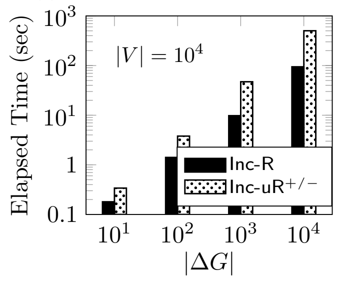
(e) Time *w.r.t.* # Partitions



(f) Vertical/Horizontal Partition



(g) Varying $|V|$ on Syn



(h) Varying $|\Delta G|$ on Syn

Figure 4: Computational Speedup on Real and Synthetic Datasets

DBLP is a co-authorship graph. Based on the collaboration time, we extracted 5 snapshots. The dataset has 391,446,225 pairs of authors (nodes) and 103,791 papers (edges).

HepPh is a citation digraph from the e-print arXiv and covers all the citations within a dataset of 1,193,426,116 pairs of papers (nodes) and 421,578 citations (edges).

Wiki contains voting data from the inception of Wikipedia, where an edge is a vote from a user to another. The dataset has 68,840,209 pairs of users (nodes) and 103,689 votes (edges).

Email is an Email network of a EU research institution, where a node is an email address, and an edge $i \to j$ is a message from i to j. The dataset contains 70,338,465,796 pairs of email addresses and 420,045 messages (edges).

2) Synthetic Datasets. RTG (Random Typing Generator) [1] is used to generate dynamic graphs and updates (a set of insertions/deletions). Graphs are controlled by a) the number of nodes $|V|$, and b) the number of edges $|E|$, which follows the densification power law and linkage generation models. Updates are controlled by a) update type (insertion or deletion), and b) the size of updates $|\Delta G|$.

3) Algorithms. We implement all the methods in VC 2015.

Algorithm	Description
Inc-R$^+$, Inc-R$^-$, Inc-R	our bulk updates in Algorithms 3, 4, 5
Inc-uR$^+$, Inc-uR$^-$	our unit update in Algorithms 1, 2
Bear [8]	LU decomposition + block elimination
k-dash [3]	LU decomposition + tree estimation
MC [2]	Monte Carlo-based incremental RWR
B-LIN [11]	graph partitioning + low-rank SVD
IRWR [14]	incremental RWR (disallow $\mathbf{A}$ size change)
DAP [10]	direction-aware RWR (for all queries)

4) Parameters. We take the following parameters by default, as previously adopted by [10], [11]: a) the decay factor $\gamma = 0.9$, b) the number of partitions for B-LIN, $\tau = 100$, and c) the total number of iterations for DAP, $K = 80$.

5) Accuracy Metrics. Two measures of accuracy are used: average difference (AD) and F-score. AD is defined as AD $:= \frac{1}{|V|^2}(\sum_{i,j} |\mathbf{P}_{i,j} - \hat{\mathbf{P}}_{i,j}|^2)^{1/2}$. It can assess the average error of proximities over all pairs by deterministic algorithms.

F-score is defined as F-score $:= 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$.

Since [3] has theoretically proved the exactness of k-dash, we can choose its proximity scores as the ideal baseline.

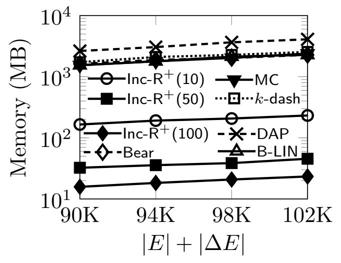
All experiments are run on an Intel Core(TM) i7-4700MQ CPU @ 2.40GHz CPU and 32GB RAM, using Windows 7.

The running time includes both CPU time and I/O costs.

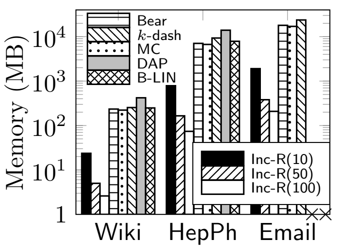
B. Experimental Results

Exp 1) Speedup. We first evaluate the running time of Inc-R$^+$ and Inc-R$^-$ on DBLP. The results are shown in Figures 4a–4b. We discern that (a) as $|E|$ increases from 90K to 102K (*resp.* decreases from 103K to 91K), Inc-R$^+$ (*resp.* Inc-R$^-$) consistently outperforms other methods, *e.g.*, as $|E| = 102$K, Inc-R$^-$ is $\sim$54.3x faster than B-LIN, $\sim$20x faster than DAP and k-dash, and $\sim$13x faster than Bear. This is because Inc-R$^+$ and Inc-R$^-$ can incrementally update only the changes to all pairs of proximities that can be obtained by the outer product of two vectors, without the need to perform any matrix decomposition (*e.g.*, LU, SVD) and matrix-matrix multiplications. (b) When $|E|$ decreases, all algorithms require less running time except Inc-R$^-$ and MC. The reason is that Inc-R$^-$ and MC update proximities by reusing previous results, whose time relies on the number of updated edges.

We next test the running time of Inc-R on real datasets, by using synthetic insertions/deletions mixed together. Due to similar trend, Figure 4c only reports $|\Delta G| = 1000$. We see that (a) on each dataset, Inc-R always performs the best, *e.g.*, on Wiki, Inc-R is 25.8x faster than B-LIN, 11.3x faster than k-dash, 9.2x faster than DAP, and 6.5x faster than Bear. This high efficiency is due to 1) the representation of all proximity changes as the outer product of two vectors, and 2) our aggregation and hashing strategies for bulk updates. (b) When the size of dataset is larger, the speedup of Inc-R relative



(a) Pure Update (DBLP)

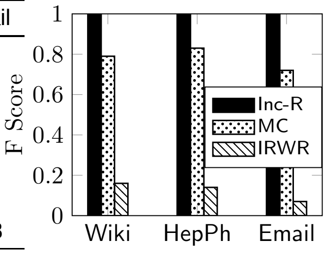


(b) Mixed Update (Real Data)

Figure 5: Memory Efficiency on Real and Synthetic Datasets

	Wiki	HepPh	Email
Inc-R	0	0	0
Bear	0	0	0
k-dash	0	0	0
DAP	0.0008	0.0006	—
B-LIN	0.0087	0.0048	—
IRWR	0.76	0.82	0.68

(a) Average Difference



(b) F-Score

Figure 6: Accuracy and Exactness

to MC is more pronounced, *e.g.,* on HepPh (*resp.* Email), Inc-R is ~7.5x (*resp.* ~14.3x) faster than MC. This is because MC is ineffective for all-pairs computation as there are redundant sampling among RWR vectors *w.r.t.* different query.

To favor IRWR that disallows new nodes created for edge updates, we rebuild all updates of case (C3) on real data, and compare IRWR with Inc-R. Figure 4d depicts the results. It can be seen that Inc-R runs consistently faster than IRWR, since Inc-R optimizes bulk updates via merging and hashing methods, whereas IRWR handles these updates one by one.

Figure 4e evaluates the effect of the number of partitions on the running time of Inc-R on real datasets. By increasing the number of partitions from 10 to 100 on each dataset, we can see that Inc-R grows slightly. This is because the growing number of partitions may lead to more I/O costs to load all-pairs proximities segment-wisely, thereby increasing the total running time. However, due to the rank-one update structure of proximity changes, after $\mathbf{P}_{i,\star}$ is memoized, our partitioning methods do not require communication costs across segments.

Figure 4f tests the impact of different partitioning methods (*e.g.,* horizontal and vertical partitioning) on the running time of Inc-R over HepPh and Email. For each dataset, we vary the number of partitions from 10 to 100, and apply horizontal and vertical partitioning, respectively, for a fixed partition size. The result shows that, given the partition size, on every dataset, the running time of Inc-R is almost the same regardless of the partitioning methods we used. This is due to the similar block structure of $\mathbf{P}$ and $\mathbf{P}^T$. Hence, the performance of the vertical partitioning is similar to that of the horizontal partitioning.

On synthetic data, we compare the running time of Inc-R for mixed bulk updates with that of multiple executions of unit update Inc-uR$^{+/-}$. In Figure 4g, we fix $|\Delta G| = 1000$ and vary $|V|$ from 10^2 to 10^5; in Figure 4h, we fix $|V| = 10^4$ and vary $|\Delta G|$ from 10^1 to 10^4. We notice that (a) Inc-R is 2.4x–7.6x faster than Inc-uR$^{+/-}$, showing the effectiveness of our aggregation approaches to minimize $\Delta G_{\min}$. (b) When $|V|$ (*resp.* $|\Delta G|$) grows, the times of both methods increase, but the speedup of Inc-R is more apparent for large $|V|$ (*resp.* $|\Delta G|$). This is because large $|\Delta G|$ and $|V|$ increase the occurrence of edge updates with a repeated end, thus enabling a huge reduction in $|\Delta G|$ after edges are sorted.

Exp 2) Memory Efficiency. Figure 5a compares the memory of all methods for pure bulk updates on DBLP. When $|\Delta E|$ increases from 90K to 102K, we notice that (a) the memory for Bear, MC, DAP, k-dash, and B-LIN stabilizes at ~2.1G. This is because these methods need store all-pairs proximities for output. In contrast, Inc-R$^+$ can incrementally update each partition with no need to load all-pairs proximities to memory. (b) When the number of partitions grows, the size of each partition for $\mathbf{P}$ becomes smaller. Thus, the memory of Inc-R$^+$ decreases, which is consistent with our analysis in Section IV.

Figure 5b shows the memory of Inc-R for the mixed bulk updates on Wiki, HepPh, and Email. Due to similar tendency, we only report the results on $|\Delta G| = 1000$. (a) On each graph, given the partition number {10,50,100}, the memory of Inc-R is less than those of other methods by 1–2 orders of magnitude. This is because, after partition, Inc-R updates each segment independently, with no need to memoize all-pairs proximities. (b) When the number of partitions increases, the memory of Inc-R decreases. (c) On large Email, B-LIN and DAP fail to allocate sufficient memory for all-pairs outputs.

Exp 3) Exactness. Figure 6 assesses the accuracy of Inc-R on Wiki, HepPh, Email by average difference and F-score. For each dataset, k-dash is selected as the baseline due to its exactness. We can see that (a) the average difference of DAP is $\sim 10^{-3}$ due to the iterative error. (b) The average difference of B-LIN is $\sim 10^{-2}$ due to its low-rank SVD approximation. (c) In all cases, the average difference of Inc-R is 0, showing the exactness of our method. (d) For IRWR, its average difference is large and F-score is small, due to the technical bugs of [14]; and Inc-R gives a full treatment. (e) The F-score of MC with 0.95 confidence is ~0.8 due to its probabilistic nature.

VII. Conclusions

In this paper, we consider the efficient computation of all pairs of RWR proximities on large dynamic graphs. Firstly, for unit update, we characterize the proximity changes as the outer product of two vectors, and observe the commutative property for RWR: $\mathbf{PA} = \mathbf{AP}$. These can substantially speed up the computation of all pairs of proximities from $O(|V|^3)$ to $O(|V|^2)$ time in the worst case, with no loss of accuracy. Then, to avoid $O(|V|^2)$ memory for all-pairs outputs, we also propose efficient partitioning methods, such that all pairs of proximities can be computed segment-wisely in only $O(l|V|)$ memory with $O(\lceil \frac{|V|}{l} \rceil)$ I/O costs, where $1 \leq l \leq |V|$ is a user-controlled trade-off between memory and I/O costs. Besides, for bulk updates, we devise aggregation and hashing methods to eliminate unnecessary updates further and handle chunks of unit updates simultaneously. Our experiments show that our method can be 10–100x faster than the best-known competitors on large graphs while securing exactness and scalability.

Acknowledgment. The work is supported by NEC Japan and the European Commission FP7 Grant (No. 619795).

References

[1] L. Akoglu and C. Faloutsos. RTG: A recursive realistic graph generator using random typing. In *PKDD*, 2009.

[2] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and Personalized PageRank. *PVLDB*, 4(3):173–184, 2010.

[3] Y. Fujiwara, M. Nakatsuji, M. Onizuka, and M. Kitsuregawa. Fast and exact top-k search for random walk with restart. *PVLDB*, 5(5), 2012.

[4] G. Jeh and J. Widom. SimRank: A measure of structural-context similarity. In *KDD*, pages 538–543, 2002.

[5] I. Konstas, V. Stathopoulos, and J. M. Jose. On social networks and collaborative recommendation. In *SIGIR*, pages 195–202, 2009.

[6] N. Lao and W. W. Cohen. Relational retrieval using a combination of path-constrained random walks. *Machine Learning*, 81(1):53–67, 2010.

[7] P. Sarkar, A. W. Moore, and A. Prakash. Fast incremental proximity search in large graphs. In *ICML*, 2008.

[8] K. Shin, J. Jung, L. Sael, and U. Kang. BEAR: Block elimination approach for random walk with restart on large graphs. In *SIGMOD*, pages 1571–1585, 2015.

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

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

[11] H. Tong, C. Faloutsos, and J. Pan. Fast random walk with restart and its applications. In *ICDM*, pages 613–622, 2006.

[12] G. Weikum and M. Theobald. From information to knowledge: Harvesting entities and relationships from web sources. In *PODS*, 2010.

[13] A. W. Yu, N. Mamoulis, and H. Su. Reverse top-k search using random walk with restart. *PVLDB*, 7(5):401–412, 2014.

[14] W. Yu and X. Lin. IRWR: Incremental random walk with restart. In *SIGIR (poster version)*, pages 1017–1020, 2013.

[15] W. Yu, X. Lin, W. Zhang, and J. A. McCann. Fast all-pairs simrank assessment on large graphs and bipartite domains. *IEEE Trans. Knowl. Data Eng.*, 27(7):1810–1823, 2015.

[16] W. Yu and J. A. McCann. Sig-SR: SimRank search over singular graphs. In *ACM SIGIR*, pages 859–862, 2014.

[17] W. Yu and J. A. McCann. Co-Simmate: Quick retrieving all pairwise Co-Simrank scores. In *ACL*, pages 327–333, 2015.

[18] W. Yu and J. A. McCann. Gauging correct relative rankings for similarity search. In *ACM CIKM*, pages 1791–1794, 2015.

[19] W. Yu and J. A. McCann. High quality graph-based similarity search. In *ACM SIGIR*, pages 83–92, 2015.

[20] F. Zhu, Y. Fang, K. C. Chang, and J. Ying. Incremental and accuracy-aware personalized PageRank through scheduled approximation. *PVLDB*, 6(6):481–492, 2013.

Appendix

Proof of Theorem 1. By RWR definition (1), we have

$$\begin{aligned}\tilde{\mathbf{P}} &= (1-\gamma)(\mathbf{I}-\gamma\tilde{\mathbf{A}})^{-1} \quad \text{(plugging Eq.(3) into } \tilde{\mathbf{A}}\text{)}\\ &= (1-\gamma)\left[\begin{array}{c|c}\mathbf{I}-\gamma\mathbf{A} & -\gamma\mathbf{e}_j\\ \hline \mathbf{0} & 1\end{array}\right]^{-1} \quad \text{(using block matrix inverse)}\\ &= (1-\gamma)\left[\begin{array}{c|c}(\mathbf{I}-\gamma\mathbf{A})^{-1} & \gamma(\mathbf{I}-\gamma\mathbf{A})^{-1}\mathbf{e}_j\\ \hline \mathbf{0} & 1\end{array}\right] = \left[\begin{array}{c|c}\mathbf{P} & \gamma\mathbf{P}_{\star,j}\\ \hline \mathbf{0} & 1-\gamma\end{array}\right] \quad \square\end{aligned}$$

Proof of Lemma 2. Eq.(1) implies $\mathbf{P} = (1-\gamma)(\mathbf{I}-\gamma\mathbf{A})^{-1}$. Since $\|\mathbf{A}\|_\infty \le 1$ and $0<\gamma<1$, we have

$$(\mathbf{I}-\gamma\mathbf{A})^{-1} = \mathbf{I}+\gamma\mathbf{A}+\gamma^2\mathbf{A}^2+\cdots$$

Substituting this back into $\mathbf{PA}$ yields

$$\begin{aligned}\mathbf{PA} &= (1-\gamma)(\mathbf{I}-\gamma\mathbf{A})^{-1}\mathbf{A} = (1-\gamma)\left(\mathbf{A}+\gamma\mathbf{A}^2+\cdots\right)\\ &= \mathbf{A}(1-\gamma)(\mathbf{I}-\gamma\mathbf{A})^{-1} = \mathbf{AP} \qquad \square\end{aligned}$$

Proof of Theorem 2. We split the proof into two cases:
(1) When $\mathcal{O}(i)=0$, the proof is similar to Theorem 1.

(2) When $\mathcal{O}(i)\neq 0$, by substituting Eq.(5) into the RWR definition $\tilde{\mathbf{P}} = (1-\gamma)(\mathbf{I}-\gamma\tilde{\mathbf{A}})^{-1}$, we have

$$\tilde{\mathbf{P}} = (1-\gamma)\left[\begin{array}{c|c}\mathbf{M} & \mathbf{0}\\ \hline -\mathbf{y}^T & 1\end{array}\right]^{-1} = (1-\gamma)\left[\begin{array}{c|c}\mathbf{M}^{-1} & \mathbf{0}\\ \hline \mathbf{y}^T\mathbf{M}^{-1} & 1\end{array}\right], \quad (15)$$

where $\mathbf{M} := \mathbf{I}-\gamma\mathbf{A}+\mathbf{A}_{\star,i}\mathbf{y}^T$ and $\mathbf{y} := \frac{\gamma}{O(i)+1}\mathbf{e}_i$.

Using Sherman-Morrison inverse formula to $\mathbf{M}^{-1}$ yields

$$\begin{aligned}\mathbf{M}^{-1} &= (\mathbf{I}-\gamma\mathbf{A})^{-1} - \tfrac{(\mathbf{I}-\gamma\mathbf{A})^{-1}\mathbf{A}_{\star,i}\mathbf{y}^T(\mathbf{I}-\gamma\mathbf{A})^{-1}}{1+\mathbf{y}^T(\mathbf{I}-\gamma\mathbf{A})^{-1}\mathbf{A}_{\star,i}}\\ &= \tfrac{1}{1-\gamma}\left(\mathbf{P}-\tfrac{\mathbf{PA}_{\star,i}\mathbf{y}^T\mathbf{P}}{1-\gamma+\mathbf{y}^T\mathbf{PA}_{\star,i}}\right). \qquad (16)\end{aligned}$$

Using Lemma 2 and Eq.(2), we have $\mathbf{PA}_{\star,i} = \mathbf{AP}_{\star,i} = \frac{1}{\gamma}\left(\mathbf{P}_{\star,i}-(1-\gamma)\mathbf{e}_i\right)$. Substitute this and $\mathbf{y} = \frac{\gamma}{O(i)+1}\mathbf{e}_i$ into $(-\frac{1}{1-\gamma}\mathbf{PA}_{\star,i}\mathbf{y}^T\mathbf{P})$ in Eq.(16), which yields

$$\begin{gathered}-\tfrac{1}{1-\gamma}\mathbf{PA}_{\star,i}\mathbf{y}^T\mathbf{P} = -\tfrac{1}{1-\gamma}\left(\tfrac{1}{\gamma}(\mathbf{P}_{\star,i}-(1-\gamma)\mathbf{e}_i)\right)\mathbf{y}^T\mathbf{P} = \mathbf{z}\cdot\mathbf{P}_{i,\star}\\ \text{with } \mathbf{z} := \tfrac{1}{(O(i)+1)(1-\gamma)}\left((1-\gamma)\mathbf{e}_i-\mathbf{P}_{\star,i}\right).\end{gathered}$$

Substituting the above equation back to Eq.(16) produces

$$(1-\gamma)\mathbf{M}^{-1} = \mathbf{P}+\tfrac{\mathbf{z}\cdot\mathbf{P}_{i,\star}}{1-\mathbf{z}_i} \quad \text{and} \quad (1-\gamma)\mathbf{y}^T\mathbf{M}^{-1} = \tfrac{\gamma}{O(i)+1}\left(\tfrac{\mathbf{P}_{i,\star}}{1-\mathbf{z}_i}\right).$$

Plugging these two equations into Eq.(15) yields Eq.(8). $\square$

Proof of Lemma 3. For $\mathcal{O}(i)=0$, $\mathbf{\Delta A} = \mathbf{e}_j\mathbf{e}_i^T$.

For $\mathcal{O}(i)\neq 0$, after insertion, there are 2 changes in $\tilde{\mathbf{A}}_{\star,i}$: (1) all the nonzeros of $\mathbf{A}_{\star,i}$ are updated from $\frac{1}{\mathcal{O}(i)}$ to $\frac{1}{\mathcal{O}(i)+1}$; (2) the j-th entry of $\mathbf{A}_{\star,i}$ is changed from 0 to $\frac{1}{\mathcal{O}(i)+1}$. Thus,

$$\tilde{\mathbf{A}}_{\star,i} = \tfrac{\mathcal{O}(i)}{\mathcal{O}(i)+1}\mathbf{A}_{\star,i}+\tfrac{1}{\mathcal{O}(i)+1}\mathbf{e}_j = \mathbf{A}_{\star,i}+\mathbf{u},$$

where $\mathbf{u} := \frac{1}{\mathcal{O}(i)+1}(\mathbf{e}_j-\mathbf{A}_{\star,i})$. Hence, Eq.(9) holds. $\square$

Proof of Lemma 4. Eq.(1) implies $\frac{1}{1-\gamma}(\mathbf{I}-\gamma\tilde{\mathbf{A}})\tilde{\mathbf{P}} = \mathbf{I}$. By Lemma 3, we plug $\tilde{\mathbf{A}} := \mathbf{A}+\mathbf{u}\mathbf{e}_i^T$ into the above equation:

$$\tfrac{1}{1-\gamma}(\mathbf{I}-\gamma\mathbf{A})\tilde{\mathbf{P}}-\mathbf{u}\mathbf{v}^T = \mathbf{I} \quad \text{with} \quad \mathbf{v}^T = \tfrac{\gamma}{1-\gamma}\tilde{\mathbf{P}}_{i,\star}.$$

In block matrix forms, these equations can be rewritten as

$$\left[\begin{array}{c|c}\frac{1}{1-\gamma}(\mathbf{I}-\gamma\mathbf{A}) & -\mathbf{u}\\ \hline \frac{\gamma}{1-\gamma}\mathbf{e}_i^T & -1\end{array}\right]\left[\begin{array}{c}\tilde{\mathbf{P}}\\ \hline \mathbf{v}^T\end{array}\right] = \left[\begin{array}{c}\mathbf{I}\\ \hline \mathbf{0}\end{array}\right].$$

By left-multiplying both sides by $\left[\begin{array}{c|c}\mathbf{I} & \mathbf{0}\\ \hline -\frac{\gamma}{1-\gamma}\mathbf{P}_{i,\star} & \mathbf{I}\end{array}\right]$, we have

$$\left[\begin{array}{c|c}\frac{1}{1-\gamma}(\mathbf{I}-\gamma\mathbf{A}) & -\mathbf{u}\\ \hline \mathbf{0} & \frac{\gamma}{1-\gamma}\mathbf{P}_{i,\star}\mathbf{u}-1\end{array}\right]\left[\begin{array}{c}\tilde{\mathbf{P}}\\ \hline \mathbf{v}^T\end{array}\right] = \left[\begin{array}{c}\mathbf{I}\\ \hline -\frac{\gamma}{1-\gamma}\mathbf{P}_{i,\star}\end{array}\right].$$

Applying $(\mathbf{I}-\gamma\mathbf{A})^{-1} = \frac{1}{1-\gamma}\mathbf{P}$ to the above equations yields

$$\tilde{\mathbf{P}} = \mathbf{Puv}^T+\mathbf{P} \quad \text{with} \quad \mathbf{v}^T = \left(\tfrac{\gamma\mathbf{P}_{i,q}}{1-\gamma-\gamma\mathbf{P}_{i,\star}\mathbf{u}}\right)\mathbf{P}_{i,\star} \qquad \square$$

Proof of Theorem 3. By Lemmas 3 and 4, we have
(1) If $\mathcal{O}(i)=0$, then $\mathbf{u}=\mathbf{e}_j$. We have $\mathbf{Pu}=\mathbf{P}_{\star,j}$.
(2) If $\mathcal{O}(i)\neq 0$, then $\mathbf{u} = \frac{1}{\mathcal{O}(i)+1}(\mathbf{e}_j-\mathbf{A}_{\star,i})$. We have

$$\mathbf{Pu} = \tfrac{1}{\mathcal{O}(i)+1}(\mathbf{P}_{\star,j}-\mathbf{PA}_{\star,i}) = \tfrac{1}{\mathcal{O}(i)+1}(\mathbf{P}_{\star,j}-\tfrac{1}{\gamma}\mathbf{P}_{\star,i}-(1-\tfrac{1}{\gamma})\mathbf{e}_i).$$

The last "=" is due to Eq.(6): $\mathbf{PA}_{\star,i} = \frac{1}{\gamma}(\mathbf{P}_{\star,i}-(1-\gamma)\mathbf{e}_i)$. Combining Eq.(10) with the resulting $\mathbf{Pu}$ yields Eq.(11). $\square$