Scaling Random Walk with Restart over Dynamic Networks

Weiren Yu
Department of Computing
Imperial College London
weiren.yu@imperial.ac.uk

Julie A. McCann
Department of Computing
Imperial College London
j.mccann@imperial.ac.uk

***Abstract*—Random Walk with Restart (RWR) is an appealing measure of proximity between nodes based on network topologies. As real graphs are becoming larger and subject to minor changes, it is rather cost-inhibitive to recompute proximities from scratch. Existing method utilizes LU decomposition and degree reordering heuristics. As a result, it yields $O(|V|^3)$ time and $O(|V|^2)$ memory to compute all ($|V|^2$) pairs of RWR proximities on a static graph. This paper proposes a dynamic scheme to assess all-pairs RWR: (1) For unit update, we characterize the changes to all-pairs RWR as the outer product of two vectors. Furthermore, we notice that the multiplication of an RWR proximity matrix and its transition matrix, unlike traditional matrix multiplication, is commutative. These can significantly reduce the computation of all-pairs RWR to $O(|V|^2)$ worst-case time for every update with no accuracy loss. In practice, the $O(|V|^2)$ time can be reduced to $O(|\Delta|)$ further, where $|\Delta|$ ($\leq |V|^2$) is the number of affected proximity elements. (2) To avoid $O(|V|^2)$ memory for all-pairs outputs, we also devise efficient partitioning techniques based on our incremental model, which can compute all pairs of proximities 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 usage and I/O costs. (3) For bulk updates, we devise aggregation and hashing methods that can discard unnecessary updates further and handle chunks of unit updates simultaneously. The experiments on many datasets verify that our approaches can be 1–2 orders of magnitude faster than other competitors while securing exactness and scalability.**

I. Introduction

With the increasing scale of the Internet, many applications are confronted with large and dynamically evolving networks. For instance, the World Wide Web today embraces more than one trillion links, 7% — 18% of which are updated fortnightly. A common task on graph data 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, with a wide range of emerging applications, such as nearest neighbor search [13], named entity disambiguation [5], collaborative filtering [4], automatic image labeling [11], and anomaly detection [9].

The success of RWR can be largely ascribed to its succinct and intuitive philosophy which revolves around random walks. Let us consider a random surfer starting from a given node x. The surfer has two options at each 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.

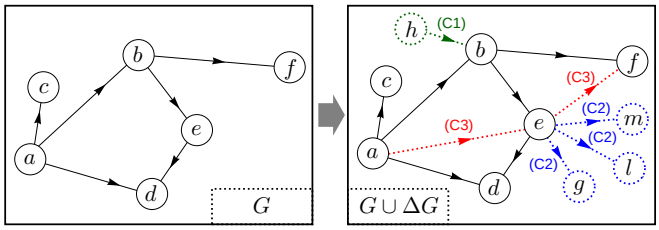


Fig. 1: An Imperial's Website Updated by 3 Types of Edge Insertions

Compared with other similarity measures (*e.g.*, SimRank), RWR highlights the following features: (1) It can recursively capture the multi-hop neighborhood information of every node. (2) Unlike SimRank which is a symmetric similarity measure for quantifying the structural equivalence between two nodes, RWR is an asymmetric measure over a directed graph, focusing on the reachability from one node to another. (3) RWR is a stable measure that is resilient to noise in a graph. Consequently, RWR has witnessed galloping attraction in fertile communities over recent years [4], [5], [9]–[13].

However, the practicability of RWR is hindered by its high computational cost. The best-of-breed methods [2], [8] use LU decomposition and degree reordering heuristics. Consequently, it entails $O(|V|^3)$ time and $O(|V|^2)$ memory in the worst case to compute all ($|V|^2$) pairs of proximities over a static graph. However, due to the dynamics and increasing scale of the Web, it is rather expensive to recompute all proximities from scratch when a graph is frequently updated with small changes.

In this paper, we consider efficient dynamic computation of all-pairs RWR proximities on large-scale evolving networks.

Given all-pairs proximities in old graph G, and updates ΔG to G (*i.e.*, a collection of new edge insertions or deletions)
Compute the changes to all-pairs RWR proximities efficiently without loss of exactness.

We are especially interested in the situation: When all-pairs proximities cannot fit into memory, can our dynamic methods compute such changes over each segment independently while securing high efficiency? Let us take the following example.

Example 1. *Figure 1 depicts a fraction of an old web graph G taken from Imperial College, where each node denotes a web page, and each edge is a hyperlink from one page to another. The old proximity matrix of all-pairs web pages was computed.*

In this new semester, the Imperial's web site is updated by adding 4 new pages and 6 new links (see the dashed edges,

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

TABLE I: Compare Inc-R with previous methods per update (where τ is the partition number, K is the iteration number, ν is the target rank of SVD, and l is a memory-I/O trade-off)

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 into memory. Thus, it is highly desirable that, in our incremental RWR method, each segment of the old proximity matrix can be updated independently (in parallel). This can greatly improve the efficiency for all-pairs proximities evaluation on dynamic graphs. □

Nonetheless, there are 2 grand challenges to dynamic RWR evaluation: (1) Due to the recursive nature of RWR, it is hard to exploit the relationship among proximity changes, ΔG, and old proximities in G. (2) It seems 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 incremental 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 computation of all $(|V|^2)$ pairs of proximities from $O(|V|^3)$ to $O(|V|^2)$ time in the worst case for every update with no loss of accuracy. In practice, the $O(|V|^2)$ time can be reduced to $O(|\Delta|)$ further, where $|\Delta|$ $(\leq |V|^2)$ is the number of affected elements in the RWR proximity matrix. The memory consumption is $O(|V|^2)$ in the worst case, which is necessary for an *in-memory* algorithm as all $(|V|^2)$ pairs of proximities need to be maintained. (Section III)

(2) To reduce its memory further, we also propose efficient partitioning techniques based on our dynamic model, in which all $(|V|^2)$ pairs proximities can be updated segment-wisely in $O(l|V|)$ 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. Recent years have witnessed growing attention to efficient RWR computation on large-scale graphs. Existing results include SVD-based RWR [11], iterative RWR [10], LU-based RWR [2], [8], reversed K-NN RWR [13], Monte Carlo RWR [1], [7], and hub length-based RWR [15].

Table I summarizes the complexity of the state-of-the-art methods for all-pairs RWR computation for every update.

Tong *et al.* [11] provided a pioneering SVD-based method, B-LIN, for RWR computation. B-LIN first divides 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. Consequently, B-LIN can well balance the performance between offline precomputation and online query.

Later, an iterative method [10] was proposed for computing RWR, namely DAP. It requires $O(K|E||V|)$ time and $O(|V|^2)$ memory to compute all-pairs proximities on a digraph $G = (V, E)$ for K iterations, with guaranteed iterative error.

Recently, Fujiwara *et al.* [2] 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. To prune unlikely top-K candidates, k-dash can incrementally update only the upper bound of RWR proximities. In contrast, our incremental approach can compute all pairs of proximities accurately in $O(|V|^2)$ worst-case time.

The existing work [1], [7] provided probabilistic methods to estimate proximities. Sarkar *et al.* [7] integrated a sampling approach with branch and bound pruning to incrementally retrieve near neighbors of one given query with high probability. Bahmani *et al.* [1] used a Monte Carlo method to incrementally estimate the proximities with probabilistic accuracy.

Zhu *et al.* [15] presented an incremental accuracy-aware way to approximate proximities. They used a hub length-based scheduling scheme to prioritize random walks. It differs from our work in that our algorithm is exact and we only use at most 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. Worse still, there are technical bugs in [14] which cannot return correct proximities. In contrast, our techniques can overcome these limitations and we also provide a corrigendum of [14] in Section III-C.

Most recently, Shin *et al.* [8] have given a fresh impetus to fast RWR computation. Their scheme, Bear, combines a block elimination approach with a Schur complement of submatrix derived from the LU decomposition, which can achieve 300x speedup further. However, their method is on static graphs.

There has also been work [13] on reverse top-K nearest neighbors search (RKNN) based on RWR proximity. Its main idea is to terminate the computation of every top-K proximity set as early as possible via an indexing strategy.

II. Preliminaries

In this section, let us 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

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

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

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

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 $\mathbf{I}$.

Throughout this paper, the computation of the entire matrix $\mathbf{P}$ is called *all-pairs RWR proximities evaluation*.

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 (in dash lines) into G. In ΔG, the insertion (h, b) pertains to case (C1); $(e, g), (e, l), (e, m)$ to case (C2); and $(a, e), (e, f)$ 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 $j \in V$. Generally, such an edge insertion may change the old transition matrix $\mathbf{A}$ into

$$\tilde{\mathbf{A}}' = \begin{array}{c|c|c} \overbrace{\mathbf{A}}^{|V| \text{ cols}} & \mathbf{0} & \overbrace{\mathbf{e}_j}^{\text{col } i} \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} & 0 \end{array} \begin{array}{l} \} \ |V| \text{ rows} \\ \\ \leftarrow \text{row } i \end{array} \tag{3}$$

For ease of representation, $\tilde{\mathbf{A}}'$ can be reduced to[1]

$$\tilde{\mathbf{A}} = \begin{bmatrix} \mathbf{A} & \mathbf{e}_j \\ \mathbf{0} & 0 \end{bmatrix} \in \mathbb{R}^{(|V|+1)\times(|V|+1)} \tag{4}$$

by removing zero rows and columns of $\tilde{\mathbf{A}}'$ (in shaded areas). Lemma 1 shows that this removal will not affect the results.

Lemma 1. *Let $\tilde{\mathbf{P}}'$ (resp. $\tilde{\mathbf{P}}$) be the RWR proximity matrix corresponding to the transition matrix $\tilde{\mathbf{A}}'$ in Eq.*(3) *(resp. $\tilde{\mathbf{A}}$ in Eq.*(4)*). If $\tilde{\mathbf{A}}$ is the submatrix formed by excluding all zero rows and columns with the index $x \in [|V|+1, i-1]$ from $\tilde{\mathbf{A}}'$ (i.e., excluding the shaded regions in Eq.*(3)*), then $\tilde{\mathbf{P}}$ can be obtained by excluding all the rows and columns with the same index $x \in [|V|+1, i-1]$ from $\tilde{\mathbf{P}}'$.*

Proof: Let $\mathbf{E}$ be an elementary matrix formed by swapping row $(|V|+1)$ and row i of the $i \times i$ identity matrix. Using elementary row/column transformations, we obtain

$$\mathbf{E} \cdot \tilde{\mathbf{A}}' \cdot \mathbf{E} = \begin{bmatrix} \tilde{\mathbf{A}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}. \tag{5}$$

[1] In Eq.(4), though the last row of $\tilde{\mathbf{A}}$ is $\mathbf{0}$, we need not replace $\mathbf{0}$ by $\frac{\mathbf{1}^T}{|V|+1}$, where $\mathbf{1}^T$ is a row vector of all 1s. This is because, unlike the existence of PageRank that is ensured by $\tilde{\mathbf{A}}$ irreduciblity, the existence of RWR is ensured by $(\mathbf{I} - \gamma\tilde{\mathbf{A}})$ invertibility, *i.e.*, whether $(\mathbf{I} - \gamma\tilde{\mathbf{A}})$ is diagonally dominant.

Notice that $\mathbf{E} \cdot \mathbf{E} = \mathbf{I}$. Then, by RWR definition in Eq.(1), *i.e.*, $\tilde{\mathbf{P}}' = (1-\gamma)(\mathbf{I} - \gamma\tilde{\mathbf{A}}')^{-1}$, we have

$$\tilde{\mathbf{P}}' = (1-\gamma)\Big(\overbrace{\mathbf{E}\cdot\mathbf{E}}^{\mathbf{I}=} - \gamma \overbrace{\mathbf{E}\begin{bmatrix}\tilde{\mathbf{A}} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{E}}^{\tilde{\mathbf{A}}'=}\Big)^{-1} = (1-\gamma)\mathbf{E}\begin{bmatrix}\mathbf{I}-\gamma\tilde{\mathbf{A}} & \mathbf{0}\\ \mathbf{0} & \mathbf{I}\end{bmatrix}^{-1}\mathbf{E}.$$

Since $(\mathbf{I} - \gamma\tilde{\mathbf{A}})^{-1} = \frac{1}{1-\gamma}\tilde{\mathbf{P}}$, the above equation implies

$$\mathbf{E} \cdot \tilde{\mathbf{P}}' \cdot \mathbf{E} = \begin{bmatrix} \tilde{\mathbf{P}} & \mathbf{0} \\ \mathbf{0} & (1-\gamma)\mathbf{I} \end{bmatrix}. \tag{6}$$

Eqs.(5) and (6) imply that after $\mathbf{E}$ swaps the same rows/cols for $\tilde{\mathbf{A}}'$ and $\tilde{\mathbf{P}}'$, upper-left block $\tilde{\mathbf{A}}$ still corresponds to upper-left block $\tilde{\mathbf{P}}$. Hence, removing the shaded regions in Eq.(5) (*i.e.*, Eq.(3)) is equivalent to removing those in Eq.(6). ■

Remark 1. *Based on Lemma 1, for edge (i, j) to be inserted, we can assume, without loss of generality, that*

- *in case (C1), new node $i \notin V$ is indexed by $(|V|+1)$;*
- *in case (C2), new node $j \notin V$ is indexed by $(|V|+1)$;*
- *in case (C4), new nodes $i \notin V$ and $j \notin V$ are indexed by $(|V|+1)$ and $(|V|+2)$, respectively.* □

Using $\tilde{\mathbf{A}}$, new $\tilde{\mathbf{P}}$ can be solved in terms of old $\mathbf{P}$ as follows:

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*

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

Proof: By RWR definition in Eq.(1), we have

$$\tilde{\mathbf{P}} = (1-\gamma)(\mathbf{I} - \gamma\tilde{\mathbf{A}})^{-1}.$$

Plugging Eq.(4) into the above equation produces

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

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

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 into G, new $\tilde{\mathbf{P}}$ can be updated via Theorem 1 as*

$\tilde{\mathbf{P}} =$

	old proximity matrix $\mathbf{P}$						$\gamma\mathbf{P}_{\star,b}$
	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	$\times\gamma$ 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
h	0	0	0	0	0	0	0.100
	zero row vector $\mathbf{0}$						$1-\gamma$

□

Intuitively, after edge $(i,j)_{i\notin V, j\in V}$ is inserted into G, each block of new $\tilde{\mathbf{P}} = \left[\begin{array}{c|c} \mathbf{P} & \gamma\mathbf{P}_{\star,j} \\ \hline \mathbf{0} & 1-\gamma \end{array}\right]$ has the following meaning:

- The upper-left block $\mathbf{P}$ remains unchanged because i is a new node ($\notin V$) with no in-links. Thus, after insertion, $i \to j$ cannot be contained by every random walk from $y \in V$ to $x \in V$. That is, the inserted edge $i \to j$ has no impact on old walks tallied by $\mathbf{P}_{x,y}, \forall (x,y) \in V \times V$.
- The upper-right block $\gamma\mathbf{P}_{\star,j}$ is a scalar multiple of $\mathbf{P}_{\star,j}$ since, after insertion, random walks from i to node $x \in V$ are a concatenation of $i \to j$ and old walks from j to x:

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

  Since the out-degree of node i is 1, old walks from j to x (tallied by $\mathbf{P}_{x,j}$) can be reused with just a multiple factor to evaluate new walks from i to x (tallied by $\tilde{\mathbf{P}}_{x,i}$).
- The lower-left block is $\mathbf{0}$ as new node i ($\notin V$) has no in-links and is not reachable by node $x \in V$, *i.e.,* $\tilde{\mathbf{P}}_{i,x} = \mathbf{0}$.

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) as such an insertion will change not only the size of old transition matrix $\mathbf{A}$, but also some entries in $\mathbf{A}$, as indicated below.

Lemma 2. *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* [2]

$$\tilde{\mathbf{A}} = \left[\begin{array}{cc} \overbrace{\mathbf{A}}^{|V| \text{ cols}} & \overbrace{\mathbf{0}}^{\text{col } j} \\ \mathbf{e}_i^T & 0 \end{array}\right] \begin{array}{l} \} \ |V| \text{ rows} \\ \leftarrow \text{row } j \end{array} \quad \text{if } \mathcal{O}(i) = 0; \tag{7}$$

$$\tilde{\mathbf{A}} = \left[\begin{array}{cc} \overbrace{\mathbf{A} + \mathbf{v}\mathbf{e}_i^T}^{|V| \text{ cols}} & \overbrace{\mathbf{0}}^{\text{col } j} \\ \frac{1}{\mathcal{O}(i)+1}\mathbf{e}_i^T & 0 \end{array}\right] \begin{array}{l} \} \ |V| \text{ rows} \\ \leftarrow \text{row } j \end{array} \quad \text{if } \mathcal{O}(i) \neq 0, \tag{8}$$

where $\mathbf{v} = -\frac{1}{\mathcal{O}(i)+1}\mathbf{A}_{\star,i} \in \mathbb{R}^{|V|\times 1}$.

Proof: For $\mathcal{O}(i) = 0$, after insertion, $\mathbf{A}_{\star,i} = [\mathbf{0}] \in \mathbb{R}^{|V|\times 1}$ becomes $\tilde{\mathbf{A}}_{\star,i} = \left[\begin{smallmatrix} \mathbf{0} \\ 1 \end{smallmatrix}\right] \in \mathbb{R}^{(|V|+1)\times 1}$; and $\mathbf{A}_{\star,x}$ is changed to $\tilde{\mathbf{A}}_{\star,x} := \left[\begin{smallmatrix} \mathbf{A}_{\star,x} \\ 0 \end{smallmatrix}\right]$ $(\forall x \neq i)$. Thus, Eq.(7) holds.

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 last ($j$-th) entry of $\tilde{\mathbf{A}}_{\star,i}$ is initialized to $\frac{1}{\mathcal{O}(i)+1}$. Thus,

$$\tilde{\mathbf{A}}_{\star,i} = \left[\begin{array}{c} \frac{\mathcal{O}(i)}{\mathcal{O}(i)+1}\mathbf{A}_{\star,i} \\ \frac{1}{\mathcal{O}(i)+1} \end{array}\right] = \left[\begin{array}{c} \mathbf{A}_{\star,i} + \mathbf{v} \\ \frac{1}{\mathcal{O}(i)+1} \end{array}\right] \text{ with } \mathbf{v} := -\frac{1}{\mathcal{O}(i)+1}\mathbf{A}_{\star,i}.$$

For other columns, $\mathbf{A}_{\star,x}$ becomes $\tilde{\mathbf{A}}_{\star,x} := \left[\begin{smallmatrix} \mathbf{A}_{\star,x} \\ 0 \end{smallmatrix}\right]$ $(\forall x \neq i)$. Hence, Eq.(8) holds as well. ■

Lemma 2 indicates that for case (C2), after edge insertion, new $\tilde{\mathbf{A}}$ is a $(|V|+1) \times (|V|+1)$ matrix formed by bordering (a) old $\mathbf{A}$ when $\mathcal{O}(i) = 0$, or (b) a rank-one update of old $\mathbf{A}$, *i.e.,* $(\mathbf{A} + \mathbf{v}\mathbf{e}_i^T)$ when $\mathcal{O}(i) \neq 0$. More importantly, vector $\mathbf{v}$ can be obtained by just scaling the i-th column of $\mathbf{A}$.

[2] Recall that $\mathcal{O}(i)$ is the out-degree of node i in old graph G.

Utilizing the structure of $\tilde{\mathbf{A}}$, we next propose an efficient method that can incrementally update new $\tilde{\mathbf{P}}$ from old $\mathbf{P}$. Our idea is to convert the computation of $\tilde{\mathbf{P}}$ into solving $(\mathbf{I} - \gamma\tilde{\mathbf{A}})^{-1}$ in terms of old $\mathbf{P}$. Indeed, when we combine Lemma 2 with block matrix inverse formula, $\tilde{\mathbf{P}}$ takes the following form:

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

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 as this inverse can be computed efficiently from $(\mathbf{I} - \gamma\mathbf{A})^{-1}$ perturbed by a rank-one update $\mathbf{A}_{\star,i}\mathbf{y}^T$. However, since $(\mathbf{I} - \gamma\mathbf{A})^{-1}$ can be obtained by scaling old $\mathbf{P}$, the main 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 $\mathbf{P}$ too?*

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

Lemma 3. *For any transition matrix $\mathbf{A}$ and its corresponding proximity matrix $\mathbf{P}$, the following property holds:*

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

Proof: It follows from Eq.(1) that $\mathbf{P} = (1-\gamma)(\mathbf{I} - \gamma\mathbf{A})^{-1}$. Since $\|\mathbf{A}\|_\infty \leq 1$ 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

$$\begin{aligned} \mathbf{PA} &= (1-\gamma)(\mathbf{I} - \gamma\mathbf{A})^{-1}\mathbf{A} = (1-\gamma)\left(\mathbf{A} + \gamma\mathbf{A}^2 + \gamma^2\mathbf{A}^3 + \cdots\right) \\ &= \mathbf{A}(1-\gamma)(\mathbf{I} - \gamma\mathbf{A})^{-1} = \mathbf{AP} \end{aligned} \quad ■$$

In general, multiplication of matrices is not commutative. However, Lemma 3 shows that any RWR proximity matrix $\mathbf{P}$ can commute with its corresponding transition matrix $\mathbf{A}$. This commutative property provides us with an efficient method to compute the changes to $(\mathbf{I} - \gamma\mathbf{A})^{-1}$ only in terms of old $\mathbf{P}$. Precisely, the changes to $(\mathbf{I} - \gamma\mathbf{A})^{-1}$, as Theorem 2 will show, involve the computation of $\mathbf{PA}$, which requires a *matrix-matrix* multiplication, entailing $O(|V|^3)$ time if carried out naively. In contrast, by Lemma 3, computing $\mathbf{PA}$ can be reduced to $O(|V|^2)$ time, requiring only *matrix scaling and subtraction*. This is because Lemma 3 enables $\mathbf{PA}$ to be computed as

$$\mathbf{PA} = \mathbf{AP} = \tfrac{1}{\gamma}\left(\mathbf{P} - (1-\gamma)\mathbf{I}\right) \tag{9}$$

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

Leveraging Lemmas 2 and 3, 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 follows:*

$$\tilde{\mathbf{P}} = \left[\begin{array}{c|c} \overbrace{\mathbf{P}}^{|V| \text{ cols}} & \overbrace{\mathbf{0}}^{\text{col } j} \\ \hline \gamma\mathbf{P}_{i,\star} & 1-\gamma \end{array}\right] \begin{array}{l} \} \ |V| \text{ rows} \\ \leftarrow \text{row } j \end{array} \quad \text{if } \mathcal{O}(i) = 0; \tag{10}$$

$$\tilde{\mathbf{P}} = \left[\begin{array}{c|c} \overbrace{\mathbf{P} + \frac{\mathbf{z}\cdot\mathbf{P}_{i,\star}}{1-\mathbf{z}_i}}^{|V| \text{ cols}} & \overbrace{\mathbf{0}}^{\text{col } j} \\ \hline \frac{\gamma}{\mathcal{O}(i)+1}\left(\frac{\mathbf{P}_{i,\star}}{1-\mathbf{z}_i}\right) & 1-\gamma \end{array}\right] \begin{array}{l} \} \ |V| \text{ rows} \\ \leftarrow \text{row } j \end{array} \quad \text{if } \mathcal{O}(i) \neq 0. \tag{11}$$

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

Proof: We split the proof into two cases:

(1) When $\mathcal{O}(i) = 0$, by Lemma 2, we have $\tilde{\mathbf{A}} = \left[\begin{smallmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{e}_i^T & 0 \end{smallmatrix}\right]$. Substituting $\tilde{\mathbf{A}}$ into $\tilde{\mathbf{P}} = (1-\gamma)(\mathbf{I} - \gamma\tilde{\mathbf{A}})^{-1}$ produces

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

(2) When $\mathcal{O}(i) \neq 0$, by using $\tilde{\mathbf{A}}$ in Eq.(8), we have

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

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

Then, using the block matrix inversion formula, we have

$$\tilde{\mathbf{P}} = (1-\gamma)\left[\begin{array}{c|c} \mathbf{M}^{-1} & \mathbf{0} \\ \hline \mathbf{y}^T\mathbf{M}^{-1} & 1 \end{array}\right]. \tag{12}$$

Using Sherman-Morrison inverse formula[3] to $\mathbf{M}^{-1}$ yields

$$\mathbf{M}^{-1} = \tfrac{1}{1-\gamma}\left(\mathbf{P} - \tfrac{\mathbf{P}\mathbf{A}_{\star,i}\mathbf{y}^T\mathbf{P}}{1-\gamma+\mathbf{y}^T\mathbf{P}\mathbf{A}_{\star,i}}\right). \tag{13}$$

By plugging Eq.(9) into (13), the term $(-\frac{1}{1-\gamma}\mathbf{P}\mathbf{A}_{\star,i}\mathbf{y}^T\mathbf{P})$ in Eq.(13) can be computed as

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

Applying the above equation to Eq.(13) produces

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

Finally, substituting the above two equations into Eq.(12) yields Eq.(11) for $\mathcal{O}(i) \neq 0$. ■

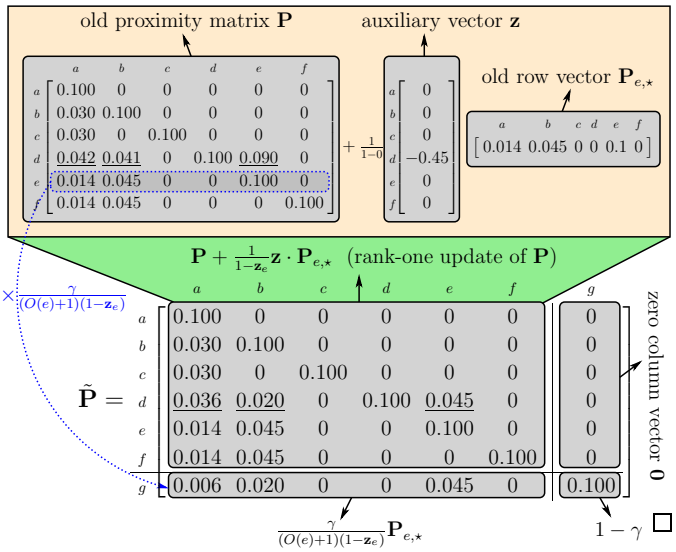
Theorem 2 gives an efficient way to incrementally compute new proximity matrix $\tilde{\mathbf{P}}$ when edge $(i,j)_{i\in V, j\notin V}$ is inserted. When $\mathcal{O}(i) \neq 0$, it requires only $O(|V|)$ time to compute $\tilde{\mathbf{P}}$ in Eq.(10), which is dominated by the computation of $\gamma\mathbf{P}_{i,\star}$. When $\mathcal{O}(i) = 0$, it requires $O(|V|^2)$ time to compute $\tilde{\mathbf{P}}$ in Eq.(11), 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 $\mathbf{P}_{i,\star}$. Thus, the total time to evaluate $\tilde{\mathbf{P}}$ by Theorem 2 is in $O(|V|^2)$, as opposed to the $O(|V|^3)$ time of the LU-based approaches [2], [8] that need evaluate $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$ to get $\tilde{\mathbf{P}}$ from scratch.

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 inserted to G, new $\tilde{\mathbf{P}}$ can be updated as follows:*

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

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

Then, noting $\mathbf{z}_e = 0$, we can obtain new $\tilde{\mathbf{P}}$ from Eq.(11):



It is worth noting that the $O(|V|^2)$ time of computing $\tilde{\mathbf{P}}$ by Eq.(11) is the *worst-case* complexity, dominated by $(\mathbf{z}\cdot\mathbf{P}_{i,\star})$. Generally, such $O(|V|^2)$ time can be reduced to $O(|\mathbf{z}||\mathbf{P}_{i,\star}|)$[4] 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 $|V|^2 = 6^2 = 36$ entries in $\mathbf{P}$.

Intuitively, when $(i,j)_{i\in V, j\notin V}$ is inserted to $G = (V,E)$, each block of new $\tilde{\mathbf{P}}$ in Theorem 2 suggests the following:

- The upper-left block of $\tilde{\mathbf{P}}$ in Eq.(10) (*resp.* Eq.(11)) keeps unchanged (*resp.* changed). The reason is that, after edge $i \to j$ ($\notin V$) is added, the increase of out-degree $\mathcal{O}(i)$ can (*resp.* cannot) alter the probability of a surfer that moves from i to any node in V when $O(i) \neq 0$ (*resp.* $O(i) = 0$), which, recursively, has an impact (*resp.* no impacts) on random walks from $\forall y \in V$ to $\forall x \in V$ tallied by $\mathbf{P}_{x,y}$.
- The upper-right block of $\tilde{\mathbf{P}}$ in Eqs.(10) — (11) is always $\mathbf{0}$ since new node j ($\notin V$) has no out-links. Thus, any node $x \in V - \{j\}$ cannot be reached from j, *i.e.*, $\tilde{\mathbf{P}}_{x,j} = \mathbf{0}$.
- The lower-left block of $\tilde{\mathbf{P}}$ in Eqs.(10) — (11) is a scalar multiple of $\mathbf{P}_{i,\star}$ since new random walks from $x \in V$ to j are a concatenation of old walks from x to i and $i \to j$:

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

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

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 $\mathbf{\Delta A} := \tilde{\mathbf{A}} - \mathbf{A}$.[5]

[3] The formula is $(\mathbf{X} + \mathbf{a}\mathbf{b}^T)^{-1} = \mathbf{X}^{-1} - \frac{\mathbf{X}^{-1}\mathbf{a}\mathbf{b}^T\mathbf{X}^{-1}}{1+\mathbf{b}^T\mathbf{X}^{-1}\mathbf{a}}$.

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

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

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

Lemma 4. *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*

$$\mathbf{\Delta A}=\mathbf{u}\mathbf{e}_i^T \text{ with } \mathbf{u}:=\begin{cases}\mathbf{e}_j & \text{if } \mathcal{O}(i)=0;\\ \frac{1}{\mathcal{O}(i)+1}(\mathbf{e}_j-\mathbf{A}_{\star,i}) & \text{if } \mathcal{O}(i)>0.\end{cases} \quad (14)$$

Proof: For $\mathcal{O}(i)=0$, $[\mathbf{\Delta A}]_{j,i}=1$. Thus, $\mathbf{\Delta A}=\mathbf{e}_j\mathbf{e}_i^T$.

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.(14) holds. ■

Lemma 4 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 inserted to old G in Figure 1, since $\mathcal{O}(a)=3$ and $\mathbf{A}_{\star,a}=[0\ \frac{1}{3}\ \frac{1}{3}\ \frac{1}{3}\ 0\ 0]^T$, it follows that*

$$\mathbf{\Delta A}=\mathbf{u}\mathbf{e}_a^T \text{ with } \mathbf{u}=\tfrac{1}{3+1}(\mathbf{e}_e-\mathbf{A}_{\star,a})=\big[\overset{a}{0}\ \overset{b}{-\tfrac{1}{12}}\ \overset{c}{-\tfrac{1}{12}}\ \overset{d}{-\tfrac{1}{12}}\ \overset{e}{\tfrac{1}{4}}\ \overset{f}{0}\big]^T.$$ □

The rank-one factorization of $\mathbf{\Delta A}$ in Lemma 4 is exploited to characterize the corresponding proximity changes $\mathbf{\Delta P}$.

Lemma 5. *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*

$$\mathbf{\Delta P}=\mathbf{P}\mathbf{u}\mathbf{v}^T \quad \text{with} \quad \mathbf{v}^T=\big(\tfrac{\gamma}{1-\gamma-\gamma\mathbf{P}_{i,\star}\mathbf{u}}\big)\mathbf{P}_{i,\star} \quad (15)$$

where vector $\mathbf{u}$ is defined by Lemma 4.

Proof: By RWR definition in Eq.(1), new $\tilde{\mathbf{P}}$ satisfies

$$\tfrac{1}{1-\gamma}(\mathbf{I}-\gamma\tilde{\mathbf{A}})\tilde{\mathbf{P}}=\mathbf{I},$$

By Lemma 4, 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

$$\tilde{\mathbf{P}}=\mathbf{P}\mathbf{u}\mathbf{v}^T+\mathbf{P} \quad \text{with} \quad \mathbf{v}^T=\big(\tfrac{\gamma\mathbf{P}_{i,g}}{1-\gamma-\gamma\mathbf{P}_{i,\star}\mathbf{u}}\big)\mathbf{P}_{i,\star}$$ ■

Lemma 5 suggests that, for case (C3), $\mathbf{\Delta P}$ is a rank-one matrix, *i.e.*, the product of vector $(\mathbf{Pu})$ and row vector $\mathbf{v}^T$, where $\mathbf{u}$ can be obtained by Eq.(14), 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 $(\mathbf{Pu})$; and (c) $O(|V|^2)$ time for the product of $(\mathbf{Pu})$ and $\mathbf{v}^T$.

To speed up the computation of $\mathbf{\Delta P}$ in Lemma 5 further, there are two noteworthy methods: (a) Once $(\mathbf{Pu})$ is computed, $(\mathbf{P}_{i,\star}\mathbf{u})$ in Eq.(15) can be obtained directly from the i-th row of the resulting $(\mathbf{Pu})$. (b) The $O(|V|^2)$ time to compute $(\mathbf{Pu})$ can be significantly reduced to $O(|V|)$ since we observe that $(\mathbf{Pu})$ can be described as a linear combination of only two old "pivot proximity vectors" $\mathbf{P}_{\star,i}$ and $\mathbf{P}_{\star,j}$:

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

To determine scalars $\square$ and $\diamondsuit$, we have the following theorem.

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:*

$$\mathbf{\Delta P}=\big(\tfrac{1}{1-\gamma-\mathbf{y}_i}\big)\mathbf{y}\mathbf{P}_{i,\star} \quad \text{with} \quad (16)$$

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

Proof: By Lemmas 4 and 5, we can obtain:

(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

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

The last "=" is due to Eq.(9): $\mathbf{P}\mathbf{A}_{\star,i}=\frac{1}{\gamma}(\mathbf{P}_{\star,i}-(1-\gamma)\mathbf{e}_i)$. Combining Eq.(15) with the resulting $\mathbf{Pu}$ yields Eq.(16). ■

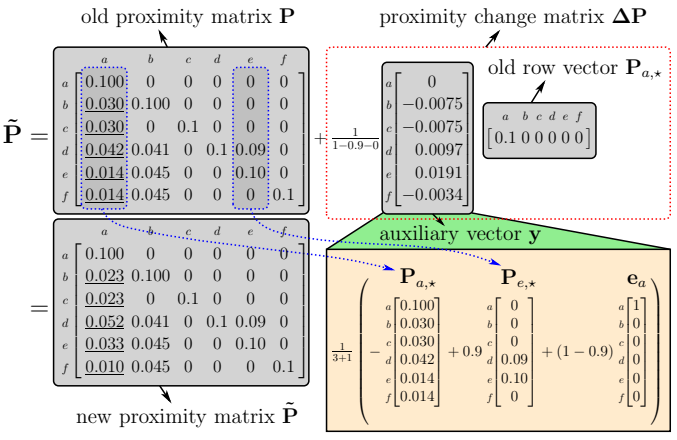
Theorem 3 is an optimized version of Lemma 5. Although the worst-case time to compute $\mathbf{\Delta P}$ by Theorem 3 is $O(|V|^2)$, the computation of $(\mathbf{Pu})$ in Lemma 5 is dramatically reduced from $O(|V|^2)$ to $O(|V|)$ time.

Moreover, the rank-one structure of $\mathbf{\Delta P}$ in Eq.(16) can also 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.(16):



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 $\mathbf{P}$. □

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 γ.
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 $\tilde{\mathbf{P}} := \begin{bmatrix} \overbrace{\mathbf{P}}^{|V| \text{ cols}} & \overbrace{\gamma \mathbf{P}_{\star,j}}^{\text{col } i} \\ \mathbf{0} & 1-\gamma \end{bmatrix}$ } $|V|$ rows; $\leftarrow$ row i
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 $\tilde{\mathbf{P}} := \begin{bmatrix} \overbrace{\mathbf{P} + \frac{\mathbf{z} \cdot \mathbf{P}_{i,\star}}{1-\mathbf{z}_i}}^{|V| \text{ cols}} & \overbrace{\mathbf{0}}^{\text{col } j} \\ \frac{\gamma}{\mathcal{O}(i)+1}\left(\frac{\mathbf{P}_{i,\star}}{1-\mathbf{z}_i}\right) & 1-\gamma \end{bmatrix}$ } $|V|$ rows; $\leftarrow$ row j
7 **else**
8 update $\tilde{\mathbf{P}} := \begin{bmatrix} \overbrace{\mathbf{P}}^{|V| \text{ cols}} & \overbrace{\mathbf{0}}^{\text{col } j} \\ \gamma \mathbf{P}_{i,\star} & 1-\gamma \end{bmatrix}$ } $|V|$ rows; $\leftarrow$ row j
9 **else if** $i \in V$ *and* $j \in V$ **then** `// Case (C3)`
10 **if** $\mathcal{O}(i) = 0$ **then**
11 set $\mathbf{y} := \gamma \mathbf{P}_{\star,j}$
12 **else**
13 set $\mathbf{y} := \frac{1}{\mathcal{O}(i)+1}(\gamma \mathbf{P}_{\star,j} - \mathbf{P}_{\star,i} + (1-\gamma)\mathbf{e}_i)$
14 update $\tilde{\mathbf{P}} := \mathbf{P} + \left(\frac{1}{1-\gamma-\mathbf{y}_i}\right)\mathbf{y}\mathbf{P}_{i,\star}$
15 **else if** $i \notin V$ *and* $j \notin V$ **then** `// Case (C4)`
16 update $\tilde{\mathbf{P}} := \begin{bmatrix} \overbrace{\mathbf{P}}^{|V| \text{ cols}} & \overbrace{\mathbf{0}}^{\text{col } i} & \overbrace{\mathbf{0}}^{\text{col } j} \\ \mathbf{0} & 1-\gamma & 0 \\ \mathbf{0} & (1-\gamma)\gamma & 1-\gamma \end{bmatrix}$ } $|V|$ rows; $\leftarrow$ row i; $\leftarrow$ row j
17 **return** $\tilde{\mathbf{P}}$

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

We next handle case (C4): the insertion of edge (i, j) with $i \notin V$ and $j \notin V$. After insertion, new transition matrix $\tilde{\mathbf{A}}$ is[6]

$$\tilde{\mathbf{A}} = \begin{bmatrix} \overbrace{\mathbf{A}}^{|V| \text{ cols}} & \overbrace{\mathbf{0}}^{\text{col } i} & \overbrace{\mathbf{0}}^{\text{col } j} \\ \mathbf{0} & 0 & 0 \\ \mathbf{0} & 1 & 0 \end{bmatrix} \begin{matrix} \} \ |V| \text{ rows} \\ \leftarrow \text{row } i \\ \leftarrow \text{row } j \end{matrix} \tag{17}$$

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*

$$\tilde{\mathbf{P}} = \begin{bmatrix} \overbrace{\mathbf{P}}^{|V| \text{ cols}} & \overbrace{\mathbf{0}}^{\text{col } i} & \overbrace{\mathbf{0}}^{\text{col } j} \\ \mathbf{0} & 1-\gamma & 0 \\ \mathbf{0} & (1-\gamma)\gamma & 1-\gamma \end{bmatrix} \begin{matrix} \} \ |V| \text{ rows} \\ \leftarrow \text{row } i \\ \leftarrow \text{row } j \end{matrix}$$

Proof: Plugging Eq.(17) to $\tilde{\mathbf{P}} = (1-\gamma)(\mathbf{I} - \gamma\tilde{\mathbf{A}})^{-1}$ yields

$$\tilde{\mathbf{P}} = (1-\gamma)\left[\begin{array}{c|c} (\mathbf{I}-\gamma\mathbf{A})^{-1} & \mathbf{0} \\ \hline \mathbf{0} & \left[\begin{smallmatrix} 1 & 0 \\ -\gamma & 1 \end{smallmatrix}\right]^{-1} \end{array}\right] = \left[\begin{array}{c|c} \mathbf{P} & \mathbf{0} \\ \hline \mathbf{0} & \left[\begin{smallmatrix} 1-\gamma & 0 \\ (1-\gamma)\gamma & 1-\gamma \end{smallmatrix}\right] \end{array}\right] \blacksquare$$

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

[6]We can assume w.l.o.g. that i and j are indexed by $(|V|+1)$ and $(|V|+2)$.

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 γ.
Output: new proximity matrix $\tilde{\mathbf{P}}$ in $G - \{(i, j)\}$.

1 **if** $\mathcal{O}(i) = 1$ **then**
2 set $\mathbf{y} := \gamma \mathbf{P}_{\star,j}$
3 **else**
4 set $\mathbf{y} := \frac{1}{\mathcal{O}(i)-1}(\gamma \mathbf{P}_{\star,j} - \mathbf{P}_{\star,i} + (1-\gamma)\mathbf{e}_i)$
5 update $\tilde{\mathbf{P}} := \mathbf{P} - \frac{1}{(1-\gamma-\mathbf{y}_i)}\mathbf{y}\mathbf{P}_{i,\star}$
6 **if** *i or j is an isolated node after deletion* **then** delete i or j
7 **return** $\tilde{\mathbf{P}}$

Note that $\tilde{\mathbf{P}}_{j,i} = (1-\gamma)\gamma$ is initialized by new edge (i, j). This value can also be used as a starting point if one wants to run the incremental algorithm starting from a singleton edge (i, j) to generate a given graph.

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. It can support all types of edge insertions over existing or new nodes. For each type, new $\tilde{\mathbf{P}}$ can be efficiently computed from old $\mathbf{P}$ without any matrix-matrix multiplications.

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 9–14), and (C4) (Lines 15–16), respectively.

For computational cost, we have the following result.

Theorem 5. *For any edge to be 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.*

Proof: The $O(|V|^2)$ time, in the worst case, is dominated by two products of vectors $\mathbf{z}\mathbf{P}_{i,\star}$ (Line 6) and $\mathbf{y}\mathbf{P}_{i,\star}$ (Line 14); the rest of the operations includes vector scaling and addition, yielding only $O(|V|)$ time. For memory usage, $O(|V|^2)$ space is used to store all pairs of old and new proximities; besides, $O(|V|)$ space is required to store intermediate vectors $\mathbf{y}, \mathbf{z}$. ■

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).

The $O(|V|^2)$ memory is necessary for an *in-memory* algorithm, due to all pairs of outputs. In Section IV, we will devise partitioning techniques to reduce the memory usage further.

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 gives a decremental method to update all-pairs proximities for unit deletion. The proofs of its correctness and complexity bounds are similar to those of Theorem 3, and are omitted here for brevity.

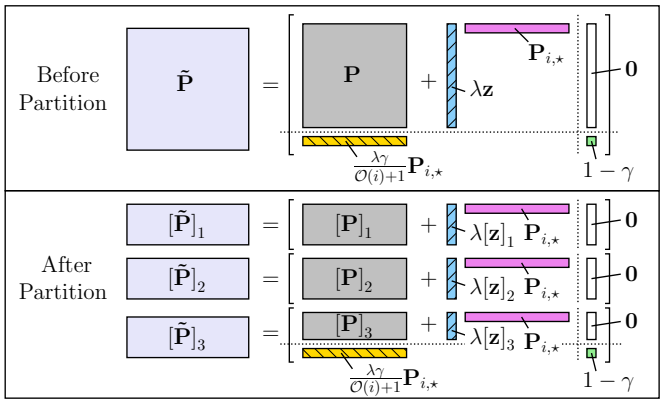


Fig. 2: Compute $\tilde{\mathbf{P}}$ in Eq.(11) Segment-wisely via Eqs.(18)–(19)

IV. Avoid Memoizing All-Pairs Proximities

In the last section, the $O(|V|^2)$ memory of our incremental method is dominated by storing all-pairs 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 mainly focus our partitioning methods on updating $\mathbf{P}$ in case (C2) for $\mathcal{O}(i) \neq 0$, *i.e.*, Eq.(11):

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

as this is the most complicated case among (C1) — (C4).

Our main 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, which might be smaller), where $1 \leq l \leq |V|$ is a user-specified integer that makes each segment small enough to fit in memory. After partitioning, $\mathbf{P}$ and $\mathbf{z}$ becomes

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

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

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

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:

$$[\tilde{\mathbf{P}}]_x = \left[\begin{array}{c|c} \overbrace{[\mathbf{P}]_x + \lambda[\mathbf{z}]_x \cdot \mathbf{P}_{i,\star}}^{|V| \text{ cols}} & \overbrace{\mathbf{0}}^{\text{col } j} \end{array}\right] \} \ l \text{ rows} \quad (\forall x = 1, \cdots, N)$$
$$\text{with } \lambda = \frac{1}{1 - \frac{1}{\mathcal{O}(i)+1}(1 - \frac{1}{1-\gamma}\mathbf{P}_{i,i})} \text{ and } [\mathbf{z}]_x = \frac{1}{\mathcal{O}(i)+1}([\mathbf{e}_i]_x - \frac{1}{1-\gamma}[\mathbf{P}_{\star,i}]_x). \quad (18)$$

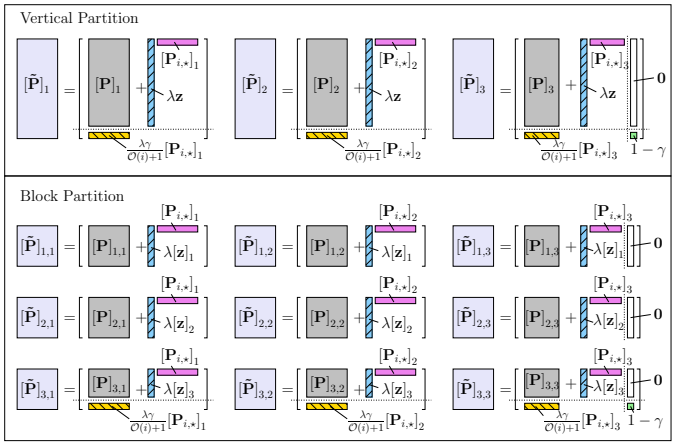


Fig. 3: Compute $\tilde{\mathbf{P}}$ in Eq.(11) via Vertical Partitioning into 3 Segments or via Block Partitioning into 3×3 Segments

except for the last segment being

$$[\tilde{\mathbf{P}}]_N = \left[\begin{array}{c|c} \overbrace{[\mathbf{P}]_N + \lambda[\mathbf{z}]_N \cdot \mathbf{P}_{i,\star}}^{|V| \text{ cols}} & \overbrace{\mathbf{0}}^{\text{col } j} \\ \hline \frac{\lambda\gamma}{\mathcal{O}(i)+1}\mathbf{P}_{i,\star} & 1-\gamma \end{array}\right] \begin{array}{l} \} \ (|V| - (N-1)l - 1) \text{ rows} \\ \leftarrow \text{row } j \end{array} \quad (19)$$

The advantage of our partitioning method in Eqs.(18)–(19) 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 $\mathbf{P}$ can be updated independently (in parallel). Figure 2 pictorially depicts how our partitioning way of Eqs.(18)–(19) segment-wisely updates $\mathbf{P}$.

The integer $1 \leq l \leq |V|$ is a user-controlled parameter that is a trade-off to balance 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 we discussed in Section III.

The CPU time for updating each segment of $\mathbf{P}$ in Eqs.(18)–(19) 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 $O(|V|^2)$ 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.[7]

In addition to the proposed horizontal partitioning method in Eqs.(18)–(19), we can similarly devise vertical partitioning and block partitioning techniques to incrementally evaluate $\mathbf{P}$. For example, as picturized in Figure 3, $\tilde{\mathbf{P}}$ in Eq.(11) can also be split via vertical partitioning and block partitioning methods. Due to a similar bordered block structure between $\tilde{\mathbf{P}}$ and $\tilde{\mathbf{P}}^T$, the performance of vertical partitioning method is similar to its horizontal counterpart, as will be validated by our experiments in Section VI as well. The block partitioning method, however, bears an extra advantage: If the memory space is rather limited ($< O(|V|)$) so that even one row/column of $\tilde{\mathbf{P}}$ cannot fit into it,

[7] Despite the same CPU time (no I/Os), the running time (with $O(\lceil \frac{|V|}{l} \rceil)$ I/Os) of our partitioning incremental algorithm will become slow when l drops.

Algorithm 3: Pure Bulk Insertions

Input : old graph $G = (V, E)$, decay factor γ,
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 ΔG 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 $J := \{\text{node } j : (i, j) \in \Delta G_{i_{\max}}\}$ 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 $\Delta G := \varnothing$
6 **return** $\tilde{\mathbf{P}}$

we can utilize the block partitioning method that can set small size $l \times l$ for each segment of $\tilde{\mathbf{P}}$ to fit into $O(l^2)$ memory.

V. Bulk Updates

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

A. Pure Bulk Insertions

Given a set of edges to be inserted into old $G = (V, E)$:

$$\Delta G := \{(i_1, j_1), (i_2, j_2), \cdots, (i_\delta, j_\delta)\},$$

where i_k and j_k $(1 \le k \le \delta)$ can be new or 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 practice, there are often many repeated nodes in ΔG. This gives us an opportunity 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: $\Delta G^1_{i_k}$ and $\Delta G^2_{i_k}$, according to whether its tail node j_k is in old 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:* $\Delta G^1_e = \{(e, f)\}$ *and* $\Delta G^2_e = \{(e, g), (e, m), (e, l)\}$. □

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:

$$\Delta G_i := \{(i, x)\}_{\forall x \in J} \quad \text{with} \quad J := \{j_1, \cdots, j_\delta\}.^8$$

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

Without loss of generality, it can be tacitly assumed that

[8] For notation convenience, we omit all the subscripts k here. Strictly, ΔG_i (*resp.* δ) should be written as $\Delta G^{1|2}_{i_k}$ (*resp.* δ_k).

Case	New Transition Matrix $\tilde{\mathbf{A}}$	New Proximity Matrix $\tilde{\mathbf{P}}$
(C1): $i \notin V$, $j_1 \in V$, $\cdots$, $j_\delta \in V$	$\begin{bmatrix} \mathbf{A} & (1/\delta)\mathbf{e}_J \\ \mathbf{0} & 0 \end{bmatrix}$ ($\mathbf{A}$: $\|V\|$ cols; $(1/\delta)\mathbf{e}_J$: col i; rows: $\|V\|$, row i) with $(\mathbf{e}_J)_x := \begin{cases} 1, & x \in J \\ 0, & x \notin J \end{cases}$	$\begin{bmatrix} \mathbf{P} & \frac{\gamma}{\delta}\sum_{j\in J}\mathbf{P}_{\star,j} \\ \mathbf{0} & 1-\gamma \end{bmatrix}$ ($\mathbf{P}$: $\|V\|$ cols; $\frac{\gamma}{\delta}\sum_{j\in J}\mathbf{P}_{\star,j}$: col i; rows: $\|V\|$, row i)
(C2): $i \in V$, $j_1 \notin V$, $\cdots$, $j_\delta \notin V$	① if $\mathcal{O}(i) = 0$, then $\tilde{\mathbf{A}} := \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ (1/\delta)\mathbf{1}_\delta\mathbf{e}_i^T & \mathbf{0} \end{bmatrix}$ ($\mathbf{A}$: $\|V\|$ cols; $\mathbf{0}$: δ cols; rows: $\|V\|$, δ) with $\mathbf{1}_\delta := [1, 1, \cdots, 1]^T$ (δ) ② if $\mathcal{O}(i) \neq 0$, then $\tilde{\mathbf{A}} := \begin{bmatrix} \mathbf{A} + \mathbf{v}\mathbf{e}_i^T & \mathbf{0} \\ \frac{1}{\mathcal{O}(i)+\delta}\mathbf{1}_\delta\mathbf{e}_i^T & \mathbf{0} \end{bmatrix}$ ($\mathbf{A}+\mathbf{v}\mathbf{e}_i^T$: $\|V\|$ cols; $\mathbf{0}$: δ cols; rows: $\|V\|$, δ) with $\mathbf{v} := -\frac{\delta}{\mathcal{O}(i)+\delta}\mathbf{A}_{\star,i}$	① if $\mathcal{O}(i) = 0$, then $\tilde{\mathbf{P}} := \begin{bmatrix} \mathbf{P} & \mathbf{0} \\ \frac{\gamma}{\delta}\mathbf{1}_\delta\mathbf{P}_{i,\star} & (1-\gamma)\mathbf{I} \end{bmatrix}$ ($\mathbf{P}$: $\|V\|$ cols; $\mathbf{0}$: δ cols; rows: $\|V\|$, δ) ② if $\mathcal{O}(i) \neq 0$, then $\tilde{\mathbf{P}} := \begin{bmatrix} \mathbf{P} + \frac{\mathbf{z}\mathbf{P}_{i,\star}}{1-\mathbf{z}_i} & \mathbf{0} \\ \frac{\gamma\mathbf{1}_\delta\mathbf{P}_{i,\star}}{(\mathcal{O}(i)+\delta)(1-\mathbf{z}_i)} & (1-\gamma)\mathbf{I} \end{bmatrix}$ ($\mathbf{P}+\frac{\mathbf{z}\mathbf{P}_{i,\star}}{1-\mathbf{z}_i}$: $\|V\|$ cols; $\mathbf{0}$: δ cols; rows: $\|V\|$, δ) 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): $i \notin V$, $j_1 \notin V$, $\cdots$, $j_\delta \notin V$	$\begin{bmatrix} \mathbf{A} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & \frac{1}{\delta}\mathbf{1}_\delta & \mathbf{0} \end{bmatrix}$ ($\mathbf{A}$: $\|V\|$ cols; col i; δ cols; rows: $\|V\|$, row i, δ)	$\begin{bmatrix} \mathbf{P} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 1-\gamma & \mathbf{0} \\ \mathbf{0} & \frac{(1-\gamma)\gamma}{\delta}\mathbf{1}_\delta & (1-\gamma)\mathbf{I} \end{bmatrix}$ ($\mathbf{P}$: $\|V\|$ cols; col i; δ cols; rows: $\|V\|$, row i, δ)

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

- in case (C1), new node $i \notin V$ is indexed by $(|V| + 1)$;
- in case (C2), new node $j_k \notin V$ is indexed by $(|V| + k)$ $(\forall k)$;
- in case (C4), new nodes $i \notin V$ and $j_k \notin V$ are indexed by $(|V| + 1)$ and $(|V| + 1 + k)$ $(\forall k)$, respectively.

Similar to unit insertion, for each case of bulk insertions, we can obtain new transition matrix $\tilde{\mathbf{A}}$ in response to ΔG_i, as depicted in the second column of Table II. We can see that, due to the reordering of $\{(i_k, j_k)\}$ in ΔG, new $\tilde{\mathbf{A}}$ also exhibits a rank-one update structure. Utilizing $\tilde{\mathbf{A}}$, we can evaluate new $\tilde{\mathbf{P}}$ by generalizing Theorems 1–4 to bulk insertions. As shown in the last column of Table II, new $\tilde{\mathbf{P}}$ in response to $\Delta G_i = \{(i, x)\}_{\forall x \in J}$ also bears a rank-one update structure.

Table II provides an incremental algorithm to compute $\tilde{\mathbf{P}}$ for pure bulk insertions, as shown in Algorithm 3. The *actual* computational cost of Algorithm 3 is hard to analyze since it relies on input parameter ΔG, *i.e.*, the type of inserted edges. However, among the 4 cases in Table II, as the most expensive computational cost is dominated by cases (C2) and (C3), the *worst-case* complexity of Algorithm 3 can be 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). Then, Algorithm 3 entails, in the worst case,* $O(|I||\tilde{V}|^2 + |\Delta G||\tilde{V}|)$ *time and* $O(|\tilde{V}|^2)$ *memory for* ΔG *bulk insertions.*

In contrast to the $O(|\Delta G||V|^2)$ worst-case time of repeated execution of Algorithm 1 for ΔG edge insertions to update $\tilde{\mathbf{P}}$, Theorem 6 shows that our bulk update method by Algorithm 3 is more efficient since $|I| \le |\Delta G|$, and in general, $|I| \ll |\Delta G|$. It is worth noticing that, the *actual* running time of Algorithm 3 is typically much faster than its worst-case time illustrated by Theorem 6, which is due to the two reasons: (a) The type of edge insertions, in practice, may not *always* happen to 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)

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 $G - \Delta G$.

sort all edges $\{(i_k, j_k)\}$ of ΔG by i_k into $|I|$ groups: $\{\Delta G_i\}$.
foreach *group* ΔG_i *in* ΔG **do**
  set $J := \{\text{node } j : (i, j) \in \Delta G_i\}$ and $\delta := |J|$.
  if $\mathcal{O}(i) = 1$ **then** $\mathbf{y} := -\frac{\gamma}{\delta}\sum_{j\in J}\mathbf{P}_{\star,j}$
  else $\mathbf{y} := \frac{1}{\mathcal{O}(i)-\delta}(\delta\mathbf{P}_{\star,i} - \gamma\sum_{j\in J}\mathbf{P}_{\star,j} - \delta(1-\gamma)\mathbf{e}_i)$
  update $\tilde{\mathbf{P}} := \mathbf{P} + \frac{1}{(1-\gamma-\mathbf{y}_i)}\mathbf{y}\mathbf{P}_{i,\star}$.
  if *i or j_1 or $\cdots$ or j_δ is an isolated node* **then**
    delete node i or j_1 or $\cdots$ or j_δ.
return $\tilde{\mathbf{P}}$

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 $\mathbf{y}$, $\mathbf{z}$, and $\mathbf{P}_{i,\star}$.

The $O(|\tilde{V}|^2)$ memory for bulk updates is necessary to an *in-memory* algorithm for all $(|\tilde{V}|^2)$ pairs output. Nevertheless, Algorithm 3 can be integrated with our partitioning methods in Section IV as well, which allows $O(\lceil\frac{|V|}{l}\rceil)$ I/O costs to reduce the memory to $O(l|\tilde{V}|)$, with $1 \le l \le |\tilde{V}|$ tuned by users.

B. 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 obtain new $\tilde{\mathbf{P}}$, unlike bulk insertions that need split edge types into 4 cases, we just need to consider one case: the deletion of a set of edges $\Delta G_i := \{(i, j_1), \cdots, (i, j_\delta)\}$ with $i \in V, j_1 \in V, \cdots, j_\delta \in V$ from old graph $G = (V, E)$. This is because, if i or j_1 or $\cdots$ or j_k is an isolated node after deletions, we can remove it later.

Algorithm 4 depicts an efficient method for bulk deletions. Its complexity is the same as Algorithm 3 (replace $|\tilde{V}|$ by $|V|$).

C. Mixed Bulk Insertions and Deletions

For mixed bulk updates, we can eliminate from ΔG many unnecessary updates that may "cancel" each other. Our main idea is to obtain a *net* update set $\Delta G_{\min}$ by using hash table to count occurrences of entries (updates) in ΔG. More 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) occurs in ΔG. Lastly, all hash keys with nonzero counts in ΔG make $\Delta G_{\min}$.

Having obtained $\Delta G_{\min}$, we sort all edges in $\Delta G_{\min}$ by its update type into two groups: net deletions $\Delta G^-_{\min}$ and net insertions $\Delta G^+_{\min}$. Then, we first invoke Pure Bulk Deletions (Algorithm 4) to update all proximities in response to changes $\Delta G^-_{\min}$, and next[9] invoke Pure Bulk Insertions (Algorithm 3) in response to changes $\Delta G^+_{\min}$. Finally, we can obtain new $\tilde{\mathbf{P}}$ in $G \oplus \Delta G_{\min}$ $(= G \oplus \Delta G$ yet $|\Delta G_{\min}| \le |\Delta G|)$.

The above process implies an efficient algorithm for mixed bulk updates, as illustrated in Algorithm 5. One can readily verify that (a) it can correctly compute new $\tilde{\mathbf{P}}$ in $G \oplus \Delta G$; (b) its computational cost, in the worst case, can be bounded by $O(|I_{\max}||\tilde{V}|^2 + |\Delta G_{\min}||\tilde{V}|)$ time and $O(|\tilde{V}|^2)$ memory,

[9]We deal with $\Delta G^-_{\min}$ *before* $\Delta G^+_{\min}$ because bulk deletions can minimize the size of the existing graph for bulk insertions.

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 $G \oplus \Delta G$.

obtain a set of net updates $\Delta G_{\min}$ from ΔG via hashing
divide $\Delta G_{\min}$ by update type into $\Delta G^-_{\min}$ and $\Delta G^+_{\min}$
call Pure Bulk Deletions (Alg. 4) to update $\tilde{\mathbf{P}}$ *w.r.t.* $\Delta G^-_{\min}$
call Pure Bulk Insertions (Alg. 3) to update $\tilde{\mathbf{P}}$ *w.r.t.* $\Delta G^+_{\min}$
return $\tilde{\mathbf{P}}$

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 $G \cup \Delta G^+_{\min}$; and (c) its memory can be reduced to $O(l|\tilde{V}|)$ with $O(\lceil\frac{|\tilde{V}|}{l}\rceil)$ I/O costs, where $1 \le l \le |\tilde{V}|$ is tuned by users.

VI. Experiments

The experimental evaluations on real and synthetic datasets verify the high efficiency of our incremental RWR methods, including its running time (with I/Os), memory, and exactness.

A. Experimental Settings

1) Real Datasets. We use 4 real datasets, including 2 temporal graphs (DBLP[10], HepPh), and 2 static graphs (Wiki, Email)[11] with synthetic updates simulating real-world evolutions.

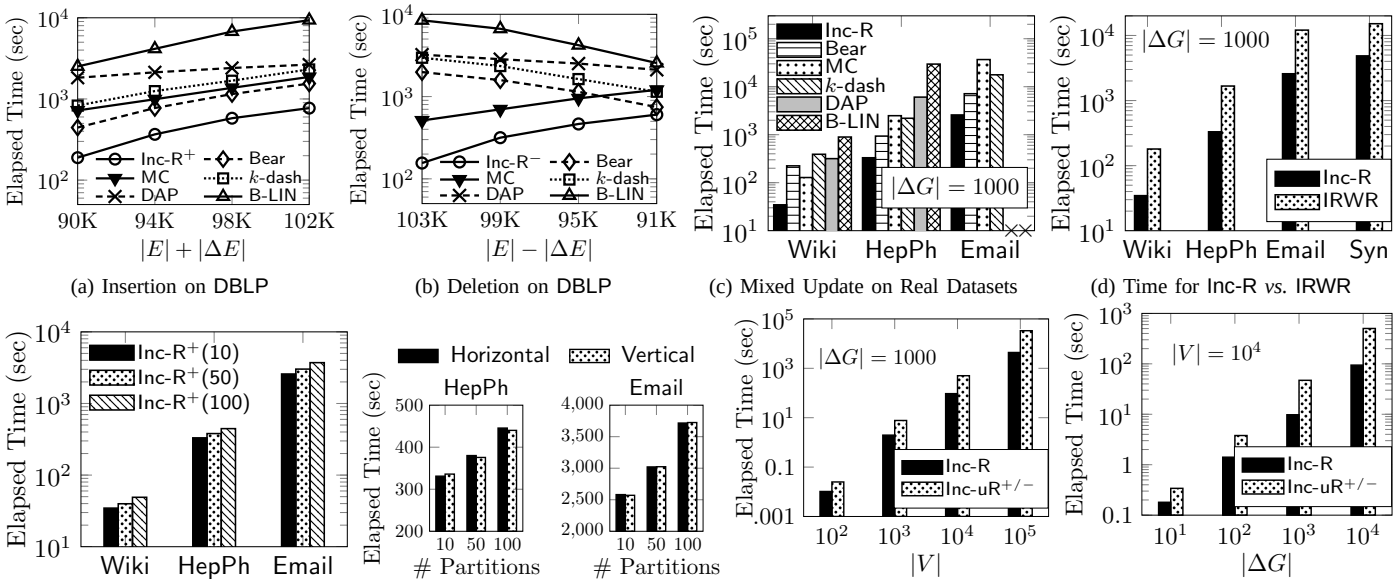
- DBLP is a co-authorship graph, where an edge exists if there is a paper collaboration between authors (nodes) in a given period of time. Based on the collaboration time interval, we extracted 5 snapshots, each with 4K edges. The entire dataset has 103K edges and 19K nodes.
- HepPh (high energy physics phenomenology) is a citation digraph from the e-print arXiv and covers all the citations within a dataset of 421,578 citations with 34,546 papers. If a paper i cites j, the graph contains an edge from i to j.
- Wiki contains voting data from the inception of Wikipedia, in which an edge is a vote from one user (node) to another. This dataset has 103,689 edges and 8,297 nodes.
- Email is an Email network of a EU research institution, where a node is an email address, and an edge $i \to j$ denotes i sent a message to j. It has 420K edges and 265K nodes.

2) Synthetic Datasets. We use Boost toolkit[12] to generate old graphs, and devise a synthetic generator to produce updates (a set of new 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 [6], and linkage generation models [3]. Updates are controlled by a) update type (insertion or deletion), and b) the size of updates $|\Delta G|$.

3) Algorithms. We implement all the algorithms in VC++.

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

[10]http://dblp.uni-trier.de/xml/
[11]http://snap.stanford.edu/data/index.html
[12]http://www.boost.org/



(a) Insertion on DBLP (b) Deletion on DBLP (c) Mixed Update on Real Datasets (d) Time for Inc-R *vs.* IRWR

(e) Time *w.r.t.* # Partitions (f) Vertical & Horizontal Partitioning (g) Varying $|V|$ on Syn (h) Varying $|\Delta G|$ on Syn

Fig. 4: Computational Speedup on Real and Synthetic Datasets

4) Parameters. We take the following parameters by default, as previously used in [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. To evaluate accuracy, two measures are used: average difference (AD) and F-score. AD is defined as $\text{AD} := \frac{1}{|V|^2}(\sum_{i,j} |\mathbf{P}_{i,j} - \hat{\mathbf{P}}_{i,j}|^2)^{1/2}$. It can assess the average error of proximities over all pairs by deterministic algorithms.

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

Since [2] 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

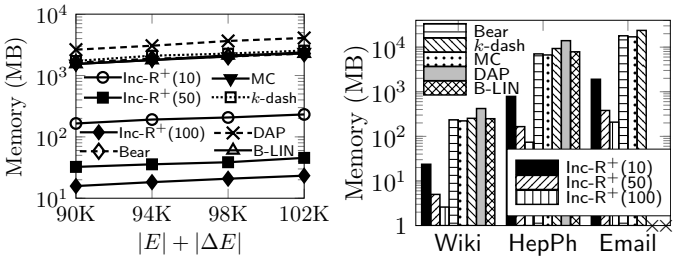
1) Speedup: We first evaluate the running time of Inc-R$^+$ and Inc-R$^-$ on DBLP. The results are shown in Figures 4a–4b. We can discern that (a) when the number of edges $|E|$ increases from 90K to 102K (*resp.* decreases from 103K to 91K), Inc-R$^+$ (*resp.* Inc-R$^-$) consistently outperforms all other methods, *e.g.*, when $|E| = 102$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 from only 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 mainly depends on the number of edges to be updated. In contrast, batch algorithms compute proximities from scratch, whose time relies on the total number of edges.

We next test the running time of Inc-R on Wiki, HepPh, Email, by using synthetic insertions/deletions mixed together. Due to similar tendency, Figure 4c only reports $|\Delta G| = 1000$. We can see that (a) on each dataset, Inc-R always outperforms all other methods, *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) our characterization of all proximity changes as the outer product of two vectors, and 2) our aggregation and hashing strategies for bulk updates. (b) When the scale of dataset becomes larger, the speedup of Inc-R relative to MC is more pronounced, *e.g.*, on HepPh (*resp.* Email), Inc-R is $\sim$7.5x (*resp.* $\sim$14.3x) faster than MC. This is because MC is ineffective for all-pairs computation as it cannot eliminate repeatedly sampling among RWR vectors *w.r.t.* different query nodes. In contrast, Inc-R can identify proximity changes as a rank-one update matrix.

To favor IRWR that disallows new nodes created for edge updates, we also 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.

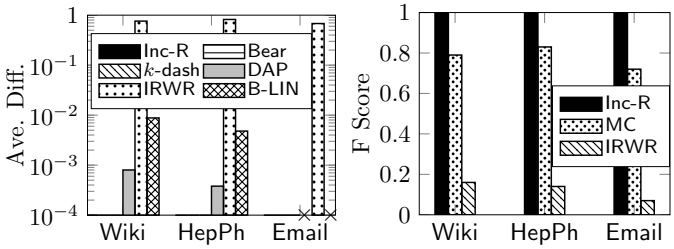
Figure 4e evaluates the effect of the number of partitions on the running time of Inc-R over Wiki, HepPh, and Email. By increasing the number of partitions from 10 to 100 on each dataset, we can see that Inc-R grows slightly. This is because the growing number of partitions may lead to more I/O costs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

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



(a) Pure Update on DBLP (b) Mixed Update on Real Datasets

Fig. 5: Memory Efficiency on Real and Synthetic Datasets



(a) Average Difference (b) F-Score

Fig. 6: Accuracy and Exactness

partitioning is similar to that of the horizontal partitioning.

Using synthetic data, we also 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 net updates $\Delta G_{\min}$. (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 $|V|$ might increase the occurrence of edge updates with a repeated end, thus enabling a huge reduction in $|\Delta G|$ after edges are sorted.

2) Memory Efficiency: Figure 5a compares the memory of all the methods for pure bulk updates on DBLP. When $|\Delta E|$ increases from 90K to 102K, we can notice that (a) the memory for Bear, MC, DAP, k-dash, and B-LIN stabilizes at $\sim$2.1G. This is because these methods need store all-pairs proximities for output. In contrast, Inc-R$^+$, given the number of partitions $\{10, 50,100\}$, can incrementally update each partition without the need to load all-pairs proximities into memory. (b) When the number of partitions increases, the size of each partition for $\mathbf{P}$ becomes smaller. Thus, the memory of Inc-R$^+$ dramatically 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 can update each segment independently, with no need to load all-pairs proximities into memory, as opposed to other methods that need store all-pairs proximities. (b) When the number of partitions increases, the memory of Inc-R decreases, as expected. (c) On large Email, B-LIN and DAP fail to allocate sufficient memory to maintain intermediate results and all-pairs outputs.

3) Exactness: Figure 6 assesses the accuracy of Inc-R on Wiki, HepPh, Email by two measures (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 $\sim 10^{-3}$, which is due to the iterative error. (b) The average difference of B-LIN is $\sim 10^{-2}$ because of its low-rank SVD approximation. (c) In all cases, the average difference of Inc-R is 0, showing the exactness of our algorithm. (d) The average difference and F-score of IRWR are large, due to the technical bugs of [14]; and Inc-R provides a full treatment. (e) The F-score of MC with 0.95 confidence interval is $\sim$0.8, due to its probabilistic nature, whereas the F-score of Inc-R is 1.

VII. Conclusions

In this paper, we consider efficient computation of all-pairs RWR proximities on 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 based on our dynamic model, 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 usage 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 experimental results on real and synthetic datasets demonstrate that our method can be 10–100x faster than the best-known competitors on large graphs while guaranteeing exactness and scalability.

References

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

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

[3] S. Garg, T. Gupta, N. Carlsson, and A. Mahanti. Evolution of an online social aggregation network: An empirical study. IMC, 2009.

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

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

[6] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM TKDD*, 1(1), 2007.

[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] 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.