Dynamic PageRank: Algorithms and Lower Bounds

Rajesh Jayaram
Google Research, New York, NY, USA

Jakub Łącki
Google Research, New York, NY, USA

Slobodan Mitrović
University of California Davis, CA, USA

Krzysztof Onak
Boston University, USA

Piotr Sankowski
IDEAS NCBR, University of Warsaw, Poland
MIM Solutions, Warsaw, Poland

Abstract

We consider the PageRank problem in the dynamic setting, where the goal is to explicitly maintain an approximate PageRank vector $\pi \in \mathbb{R}^n$ for a graph under a sequence of edge insertions and deletions. Our main result is a complete characterization of the complexity of dynamic PageRank maintenance for both multiplicative and additive (L_1) approximations.

First, we establish matching lower and upper bounds for maintaining additive approximate PageRank in both incremental and decremental settings. In particular, we demonstrate that in the worst-case $(1/\alpha)^{\Theta(\log \log n)}$ update time is necessary and sufficient for this problem, where α is the desired additive approximation. On the other hand, we demonstrate that the commonly employed ForwardPush approach performs substantially worse than this optimal runtime. Specifically, we show that ForwardPush requires $\Omega(n^{1-\delta})$ time per update on average, for any $\delta > 0$, even in the incremental setting.

For multiplicative approximations, however, we demonstrate that the situation is significantly more challenging. Specifically, we prove that any algorithm that explicitly maintains a constant factor multiplicative approximation of the PageRank vector of a directed graph must have amortized update time $\Omega(n^{1-\delta})$, for any $\delta > 0$, even in the incremental setting, thereby resolving a 13-year old open question of Bahmani et al. (VLDB 2010). This sharply contrasts with the undirected setting, where we show that poly $\log n$ update time is feasible, even in the fully dynamic setting under oblivious adversary.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms; Information systems → Page and site ranking; Theory of computation → Random walks and Markov chains; Mathematics of computing → Graph algorithms

Keywords and phrases PageRank, dynamic algorithms, graph algorithms

Digital Object Identifier 10.4230/LIPIcs.ICALP.2024.90

Category Track A: Algorithms, Complexity and Games

Funding *Slobodan Mitrović*: Supported by the Google Research Scholar and NSF Faculty Early Career Development Program #2340048.
Piotr Sankowski: Partially supported by the National Science Center (NCN) grant no. 2020/37/B/ST6/04179 and the ERC CoG grant TUgbOAT no 772346.

1 Introduction

The notion of PageRank was introduced by Brin and Page 25 years ago to rank web search results [7]. Since then, computing the PageRank of a network has become a fundamental task in data mining [23]. At a high level, PageRank is a probability distribution over the vertices

51st International Colloquium on Automata, Languages, and Programming (ICALP 2024).
Editors: Karl Bringmann, Martin Grohe, Gabriele Puppis, and Ola Svensson;
Article No. 90; pp. 90:1–90:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of a directed graph which assigns higher probability to more "central" vertices; see Section 2 for a formal definition. We write $\pi \in \mathbb{R}^n$ to denote PageRank probability vector, where π_i is the probability mass on the i-th vertex. Due to its importance, it has been studied extensively in a number of computational models. In this paper, we consider the PageRank problem in the dynamic setting, in which the goal is to maintain an approximate PageRank vector $\tilde{\pi} \in \mathbb{R}^n$ of a graph undergoing a sequence of edge insertions and deletions. We focus primarily on explicit maintainance of the PageRanks, meaning that the algorithm explicitly maintains $\tilde{\pi}$ in its memory contents at all time steps; we remark that all prior algoriths for the problem of maintaining all PageRanks in the dynamic setting have been of this form.

We consider *three* different settings, which differ in the allowed sets of operations. In the *incremental* setting, edges can only be added to the graph. Analogously, in the *decremental* setting, edges can only be deleted. The most general setting is the *fully dynamic* setting in which we allow both types of updates. We also consider two notions of approximation. A $1 + \alpha$ *multiplicative approximation* to the PageRank vector π is a vector $\tilde{\pi}$, such that for every vertex v it holds $\tilde{\pi}_v \in [(1 - \alpha)\pi_v, (1 + \alpha)\pi_v]$. An *additive* α *approximation* is a vector $\tilde{\pi}$ such that $\|\tilde{\pi} - \pi\|_1 \leq \alpha$.[1] We note that a multiplicative guarantee is strictly stronger, as a multiplicative $1 + \alpha$ approximation implies an additive α approximation.

Previous work on dynamic PageRank [4, 11, 24, 5, 10] resulted in two main approaches to the problem. The first one is based on sampling random walks. Specifically, it is well-known that one can approximate PageRank by sampling $O(\log n)$ random walks of length $O(\log n)$ from each vertex in the graph (see Algorithm 1).

In a seminal paper, Bahmani et al. [4] showed that this approach can be made dynamic. Specifically, the algorithm of Bahmani et al. maintains a *multiplicative* $1 + \alpha$ approximation of incremental (or decremental) PageRank when the updates *arrive in a random order*. However, their analysis crucially relies on the random arrival of updates, and it was not clear whether this assumption could be removed. The authors of [4] explicitly posed the question of whether it is possible to extend their results for multiplicative approximations to the case of adversarially ordered updates; to date, this question has remained open.

The second approach to computing dynamic PageRank is a dynamic version of the ForwardPush algorithm [25, 1, 9], which is a variant of a classical local push approach proposed by [3]. This algorithm was developed for the problem of maintaining Personalized PageRank, but can also be naturally used to maintain an additive PageRank approximation. While this approach is highly effective in practice, no running time bounds faster than running a static algorithm from scratch after each update have been developed for maintaining PageRank using the dynamic ForwardPush method.[2]

Thus, despite the above line of work, many fundamental questions regarding the computational cost of maintaining PageRank in a dynamic setting remain open. Specifically, it is still open whether there exists an algorithm for maintaining a approximation to PageRank in $o(n)$ time per update. This question is open even if one considers only incremental or decremental updates, or if one allows additive approximation. In this paper, we answer each of these open questions. More precisely, we characterize the complexity of solving the dynamic PageRank problem in each of these settings by providing new upper and lower bounds.

[1] Note that this coincides with the total variational distance between distributions.

[2] We note that the paper introducing the dynamic ForwardPush algorithm gives a good running time bound for running the algorithm in *undirected* graphs. However, this bound only holds for computing Personalized PageRank from a *uniformly random* source vertex. Even though PageRank can be reduced to Personalized PageRank, the reduction requires computing Personalized PageRank from a *fixed* vertex, and so the bound does not carry over.

1.1 Our contributions

We provide new lower and upper bounds on the complexity of explicitly maintaining an approximate PageRank vector both under additive and multiplicative approximation. Throughout this section, we use n to denote the number of vertices in a graph, m to denote the number of edges and ϵ to denote the jumping probability used to define PageRank.[3]

1.1.1 Additive Approximation

We provide (essentially) matching lower and upper bounds for explicitly maintaining additive approximation of PageRank in both incremental and decremental setting.

▶ **Theorem 1.** *Fix $\epsilon \in (0.01, 0.99)$. For any sufficiently large $n \geq 1$ and any α such that $1/\alpha = n^{o(1/\log\log n)}$, any algorithm which explicitly maintains α-additive approximation of PageRank must run in $n \cdot (1/\alpha)^{\Omega(\log\log n)}$ total time.*

Our lower bound, which we prove in Section 3.1, is obtained by constructing a graph and an update sequence for which the PageRank vector undergoes a large number of significant changes. The changes to the vector are large to the point that even an approximate PageRank vector must be often updated in linear time. We note that the lower bound, and all other lower bounds that we state, applies to the setting when the PageRank vector is maintained explicitly, i.e., after each update algorithm outputs the changes that the PageRank vector undergoes.

We note that it is easy to come up with an example in which a single edge update significantly changes the PageRanks of a large fraction of vertices (see Figure 1). This immediately rules out efficient incremental and decremental algorithms that maintain approximate PageRank with *worst-case* update time guarantees. This also rules out fully dynamic algorithms with amortized update time guarantees. However, proving a strong lower bound for the *amortized* update time bound in the incremental or decremental setting is far more involved, as it requires showing a long sequence of updates in which, on average, every edge insertion (or deletion) changes the PageRank of many vertices.

We complement our lower bound with the following algorithmic result proved in Section 5.

▶ **Theorem 2.** *For any $\epsilon \in (0, 1)$, there is an algorithm that with high probability explicitly maintains an α additive approximation of PageRank of any graph G in either incremental or decremental setting. The algorithm processes the entire sequence of updates in $O(m) + n \cdot (1/\alpha)^{O_\epsilon(\log\log n)}$ total time and works correctly against an oblivious adversary.*

Furthermore, we study the complexity of the dynamic ForwardPush algorithm [25]. This algorithm, when run with parameter $\tilde{\alpha}$ maintains an $\tilde{\alpha} \cdot m$ additive approximation to PageRank (and so to obtain α additive approximation, one needs to use $\tilde{\alpha} = \alpha/m$). By using a similar construction of a hard instance, we show that the algorithm takes $\Omega(n^{2-\delta})$ time, for any $\delta > 0$, to handle a sequence of $O(n)$ operations, even in incremental or decremental settings (see Theorem 9).

[3] The probability of not-jumping (in our notation, $1 - \epsilon$) is sometimes called the *damping factor* of PageRank.

1.1.2 Multiplicative Approximation

Our next result is a lower bound showing that any algorithm explicitly maintaining a constant multiplicative approximation to PageRank, even in the incremental or decremental setting, must in the worst case take $\Omega(n^{2-\delta})$ total time, for any $\delta > 0$, to process a sequence of n updates to an n-vertex graph. Specifically, we prove the following in Section 3.2:

▶ **Theorem 3.** *There exists a sequence of $\Theta(n)$ edge insertions applied to an initially empty graph on n vertices for which the following holds. For any constant $\delta > 0$, any algorithm that maintains a vector $\tilde{\pi} \in \mathbb{R}^n$ such that $(1/2)\pi_v < \tilde{\pi}_v \leq 2\pi_v$ at all time steps, must take time $\Omega(n^{2-\delta})$ to process the sequence. In particular, the amortized update time of any such algorithm is $\Omega(n^{1-\delta})$.*

We note that, by symmetry, the above theorem also applies to the decremental setting.

Theorem 3 gives a negative resolution to the 13-year-old open question of Bahmani et al. [4], who asked whether their polylogarithmic update time bounds for maintaining PageRank under a sequence of updates coming in *random* order can be extended to the general case. Previously, the only negative results for this problem were given by Lofgren [12] who showed that the specific algorithm of Bahmani et al. requires $\Omega(n^c)$ update time for some $c \in (0, 1)$, but this did not rule out the existence of a better algorithm. We extend this lower bound to *every* algorithm which explicitly maintains an approximate PageRank vector, and strengthen the bound from $\Omega(n^c)$ to $\Omega(n^{1-\delta})$ for any $\delta > 0$.

To complement the above lower bound, in Section 6, we give a simple analysis of the Bahmani et al. algorithm in *undirected* graphs, and show that in this case maintaining multiplicative approximation can be done in polylogarithmic time per update even in the fully dynamic setting. This algorithm also assumes an oblivious adversary. While the analysis is based on a simple observation, to the best of our knowledge it has not been explicitly given before.

▶ **Theorem 4.** *For any $\epsilon \in (0, 1)$, there is an algorithm that with high probability explicitly maintains a $1 + \alpha$ multiplicative approximation of PageRank of any undirected graph G in the fully dynamic setting. The algorithm handles each update in $O(\log^5 n/(\epsilon^2\alpha^2))$ time and works correctly against an oblivious adversary.*

It is an open question whether it is possible to design dynamic PageRank algorithms that bypass our lower bounds, for example, by not maintaining PageRank explicitly or looking beyond worst-case bounds and studying restricted graph classes.

1.2 Related Work

The dynamic PageRank problem has been studied in a number of recent works [4, 5, 11, 24, 8, 17, 18, 13, 19, 20, 9] studying both the theoretical and empirical aspects of the problem. One line of study considered the incremental and decremental settings with updates performed in *random* order [4, 24] and obtained algorithms that achieve $O(\log n/\epsilon)$ update time. The result of [24] is applicable in a non-random order as well, although in that case it requires $\Omega(d_v)$ running time per update done on a vertex v of degree d_v. Bahmani et al. [5] analyze their algorithm in a random graph model in which high PageRank vertices are more likely to receive new neighbors. We note that attempts at designing faster algorithms have been undertaken in [11] as well as [24]. However, these algorithms come with no provable approximation guarantees.

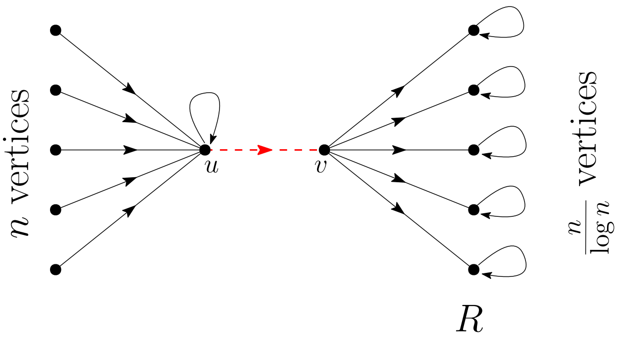

Figure 1 An example illustrating that maintaining multiplicative approximation or even an L_1 approximation of PageRank in the worst case requires $\Theta(n/\log n)$ running time even after a single deletion/insertion of edge uv. For details, see Section 1.3.

Another line of work [2, 14, 15, 25, 21] focuses on computing Personalized PageRank, which is PageRank computed from the point of view of a single vertex. For instance, [15] show that if each entry of a Personalized PageRank is lower-bounded by δ, then the Personalized PageRank of a vertex can be approximated in time $O(\sqrt{d/\delta})$, where d is the average graph degree.

Finally, PageRank was also studied in the context of sublinear algorithms [6, 22]. For instance, for a graph on m edges and omitting poly dependence on $\log m$ and α^{-1}, the very recent algorithm presented in [22] requires $O(n^{1/2} \cdot \min\{m^{1/4}, \Delta^{1/2}\})$ running time for approximating the PageRank of a single vertex, where Δ is the maximum degree in the graph.

1.3 Impossibility of Non-Trivial Worst-Case Bounds

A wealth of literature on designing dynamic algorithms for approximate PageRank, including our results, focuses on *amortized* running time complexity. It is natural to wonder whether non-trivial worst-case update running times do not exists due to lack of techniques or due to fundamental reasons. As our example in Figure 1 illustrates, non-trivial update running times are not possible even on very sparse graphs and even if one's goal is to maintain an L_1-approximate PageRank vector.

Namely, on the one hand, for the graph G in Figure 1, it can be shown that $\pi_u, \pi_v \in \Omega(\epsilon)$ and $\pi_x \in \Omega((\log n)/n)$ for each vertex $x \in R$. On the other hand, consider graph G' obtained from G, i.e., from the graph in Figure 1, by removing the red-dashed (u, v) edge, and let π' be the PageRank of G'. It is not hard to show that $\pi'_u \in \Omega(1)$, $\pi'_v \in O(\epsilon/n)$ and $\pi'_x \in O(1/n)$ for each $x \in R$. This example illustrates the following: there exists a directed graph in which after *a single* edge removal one has to update $\Omega(n/\log n)$ vertices if the goal is to maintain a multiplicative and even an L_1 approximation of the PageRank for sufficiently small constant ϵ. Moreover, if random walks are used to estimate the PageRank – which to the best of our knowledge is the only other used approach than Power method – then maintaining an additive or multiplicative approximate PageRank of a single vertex still requires $\Omega(n)$ worst-case time. To see that, observe that there are $\Theta(n)$ times more random walks passing through v in G than in G'.

Algorithm 1 An algorithm for computing approximate PageRank using random walks.
Input: A graph G, and parameters ϵ, α and ℓ.

```
Sample a set $W$ of $R = \lceil \frac{9 \ln n}{\epsilon \alpha^2} \rceil$ random walks starting from each vertex of $G$.
Each walk length is chosen from geometric distribution with parameter $1 - \epsilon$.
Remove from $W$ all walks longer than $\ell$.
for $v \in V$ do
    $\mathbf{X}_v \leftarrow$ the number of times the walks from $W$ visit $v$.
    $\tilde{\pi}(v) \leftarrow \frac{\mathbf{X}_v}{|W|/\epsilon}$.
end for
Return $\tilde{\pi}$
```

1.4 Organization of the Paper

The rest of this paper is organized as follows. In Section 2 we formally define PageRank and review a random-walk based algorithm for approximating it in the static setting. In Section 3 we give the lower bounds on the time required to explicitly maintain PageRank and on the running time of the dynamic ForwardPush algorithm. Section 4 reviews the algorithm for approximating PageRank by maintaining random walks. While the algorithm is essentially the same as the algorithm by Bahmani et al. [4], we present a full analysis, since the previous papers on dynamic PageRank did not prove the correctness of this approach. In the following two sections we analyze this algorithm in two settings. First, in Section 5 we show that this algorithm achieves near-optimal update time while maintaining additive approximation to PageRank. Second, in Section 6 we present a simple analysis showing that in undirected graphs maintaining even a constant multiplicative approximation to PageRank in the fully dynamic setting is possible with polylogarithmic update time.

2 Preliminaries

We begin by defining the PageRank of a directed graph $G = (V, E)$. Formally, the PageRank of G, denoted by $\pi \in \mathbb{R}^n_{\geq 0}$, is the stationary distribution of a random walk on G, where at each step the walk *jumps* to another uniformly random vertex with probability $\epsilon \in (0, 1)$. The jump probability ϵ is a parameter, which we will fix for the remainder. If $\deg(i)$ is the out-degree of the i-th vertex in V, then the corresponding non-symmetric transition matrix $M \in \mathbb{R}^{n \times n}$ has entries $M_{i,j} = \frac{\epsilon}{n} + (1 - \epsilon)\frac{1}{\deg(i)}$ if $(i, j) \in E$, and $M_{i,j} = \epsilon/n$ otherwise. We make the standard assumption (required for PageRank to be well-defined) that each vertex has $\deg(i) \geq 1$, which can be accomplished by adding self-loops.

PageRank can be approximated by sampling $O(\log n/\epsilon)$ relatively short random walks from each vertex. One such approach is provided as Algorithm 1, for which the following can be shown.

▶ **Proposition 5** ([4, 16]). *Let π be the PageRank vector of a graph G. The estimate $\tilde{\pi}$ computed by Algorithm 1 (with $\ell = \infty$) satisfies (a) for all $v \in V$ we have $\mathbb{E}[\tilde{\pi}_v] = \pi_v$, and (b) with probability $1 - 1/\operatorname{poly}(n)$, simultaneously for all $v \in V$, we have $\tilde{\pi}_v = (1 \pm \alpha)\pi_v$.*

3 Lower Bounds

In this section we present our lower bounds for maintaining explicit approximation to PageRank and for the running time of the dynamic ForwardPush algorithm [25]. We now describe a generic construction of a hard instance, which we instantiate with different parameters in each of the individual lower bounds. Throughout the section, we consider the case of $\epsilon \in (0.01, 0.99)$, which is the usual case in the applications of PageRank.

The Graph. The graph G is a union of the graphs H, R, S_0, S_1. First, H is a directed tree. Each non-leaf vertex v has exactly t children, with p^i parallel directed edges from v to the i-th child of v (where i is 0-based). We require that $p \geq \max(1/\epsilon, 2)$. Hence, the total-out degree of each internal vertex in H is $O(p^t)$. The depth of H is d, and so H has $\Theta(t^d)$ vertices and $\Theta(t^d \cdot p^t)$ edges.

The graph R consists of $n/4$ vertices v, each with no in-edges, and each with a single out edge vr where r is the root of the directed tree H. Finally, the sets S_0, S_1 are both directed star graphs on $s+1$ vertices (with the edges directed away from the center of the star), where S_i has the center c_i for $i \in \{0, 1\}$. Additionally, each leaf of S_0 and S_1 has a single outgoing edge, which is a self-loop. We then order the leaf vertices of H as $\ell_1, \ell_2, \dots,$ and create a directed edge from ℓ_i to $c_{i \bmod 2}$. We will set the parameters, such that the total number of vertices in H, R, S_0, and S_1 is less than n. One can then add an additional $O(n)$ isolated vertices (with self-loops), so that the total number of vertices is precisely n.

Update Sequence The initial graph has all vertices and edges of H, R, S_0, S_1, except that each non-leaf vertex of H only has an edge to its leftmost child (i.e., one with index 0). Observe that each vertex has at least one outgoing edge, and so PageRank is well-defined.

The update sequence is as follows. Let $v_1, \dots, v_{|H|}$ be the sequence of vertices visited on a pre-order traversal of H, such that $\ell_1, \ell_2, \dots$ is a subsequence of $v_1, \dots, v_{|H|}$. We insert the edges of H in $|H|$ rounds: in the ith round we insert all incoming edges of v_i (unless they have already been in the graph from the beginning).

To prove the lower bounds, we use the following way of interpreting PageRank, which is a continuous version of Algorithm 1 and follows from Proposition 5. Each vertex has some *probability mass*, which it either generates or receives from its in-neighbors. Specifically, each vertex of the graph generates a probability mass of $1/n$. A $1-\epsilon$ fraction of the probability mass of a vertex v (either generated by v or incoming to v from other vertices) is divided uniformly among the outgoing edges of v and sent to the neighbors of v. The PageRank of each vertex is exactly ϵ fraction of its probability mass.

Note that if a vertex is on a cycle, some probability mass enters it multiple times. In this case, each time the mass enters the vertex, it increases the total probability mass. In particular, we have the following.

▶ **Observation 6.** *Let v be a vertex, whose only outgoing edge is a self loop. Assume that v receives a probability mass of p along its incoming edges other than the self-loop. Then, the PageRank of v is $p + 1/n$.*

▶ **Lemma 7.** *Consider the graph G^τ obtained right after inserting all edges on the path from R to ℓ_i. Let m_i be the probability mass that reaches ℓ_i from R in G^τ. Then $m_i \geq (1-\epsilon)^{2d+2}/4$.*

Moreover, out of the probability mass that reaches the leaves of H from R, at least $(1 - 1/p)^d$ fraction reaches ℓ_i.

Proof. Observe that a path from any vertex $u \in R$ to ℓ_i first follows the edge to r, which is the only outgoing edge of u, and then, thanks to the order of adding edges of H, at each step uses the rightmost edge of each vertex in H. Consider an internal vertex $w \in H$. By the construction it has p^i edges to the ith child (0-based). Assuming that we have added edges to j children so far, we have that there are p^{j-1} edges to the rightmost child and so the fraction of outgoing edges of w that go to the rightmost child is:

$$p^{j-1} / \left(\sum_{k=0}^{j-1} p^k \right) = p^{j-1} \cdot \frac{p-1}{p^j - 1} \geq p^{j-1} \cdot \frac{p-1}{p^j} = 1 - 1/p. \tag{1}$$

The path from w to ℓ_i has $d+1$ edges. At each step $1-\epsilon$ fraction of the probability mass is forwarded to the children, out of which, as shown above, at least $1-1/p \geq 1-\epsilon$ fraction follows the path to ℓ_i. Hence, the fraction of probability mass that reaches ℓ_i from w is $(1-\epsilon)^{2d+2}$. Since vertices of R generate a total probability mass of $1/4$, we get the desired.

The second claim follows directly from Equation (1) and the fact that H has depth d. ◀

3.1 Lower Bound for Maintaining Additive Approximation

We first show the following auxiliary lemma which we will use to argue when an additive α-approximate PageRank vectors must be updated in linear time.

▶ **Lemma 8.** *Consider four vectors $v^1, \tilde{v}^1, v^2, \tilde{v}^2 \in \mathbb{R}^n$, such that $\|v^1-\tilde{v}^1\|_1 \leq \alpha$, $\|v^2-\tilde{v}^2\|_1 \leq \alpha$ and v^1 and v^2 differ by at least $100 \cdot \alpha/n$ on at least $n/4$ coordinates. Then $\tilde{v}^1$ and $\tilde{v}^2$ differ on $\Omega(n)$ coordinates.*

Proof. The proof goes by contradiction. Assume that $\tilde{v}^1$ and $\tilde{v}^2$ differ on at most $n/1000$ coordinates. Thus, they have at least $0.999 \cdot n$ coordinates in common. Moreover, $\|v^1-\tilde{v}^1\|_1 \leq \alpha$ implies that v^1, and $\tilde{v}^1$ differ by more than $10 \cdot \alpha/n$ on less than $0.1 \cdot n$ coordinates. Clearly, a similar property is satisfied by v^2, and $\tilde{v}^2$.

Let I be the set of coordinates where

1. $\tilde{v}^1$ and $\tilde{v}^2$ are equal (there are at least $0.999 \cdot n$ such coordinates),
2. v^1 and v^2 differ by at least $100 \cdot \alpha/n$ (at least $n/4$ such coordinates),
3. v^1 and $\tilde{v}^1$ differ by at most $10 \cdot \alpha/n$ (at least $0.9 \cdot n$ coordinates),
4. v^2 and $\tilde{v}^2$ differ by at most $10 \cdot \alpha/n$ (at least $0.9 \cdot n$ coordinates).

Observe that since the vectors have n coordinates, I is nonempty. By using first the triangle inequality, and then items 2-4 above, for any coordinate $i \in I$ we have

$$\begin{aligned}|\tilde{v}^1_i - \tilde{v}^2_i| &\geq |v^1_i - v^2_i| - |v^1_i - \tilde{v}^1_i| - |v^2_i - \tilde{v}^2_i| \\ &\geq 100 \cdot \alpha/n - 10 \cdot \alpha/n - 10 \cdot \alpha/n \\ &= 80 \cdot \alpha/n.\end{aligned}$$

which contradicts item 1. The lemma follows. ◀

▶ **Theorem 1.** *Fix $\epsilon \in (0.01, 0.99)$. For any sufficiently large $n \geq 1$ and any α such that $1/\alpha = n^{o(1/\log\log n)}$, any algorithm which explicitly maintains α-additive approximation of PageRank must run in $n \cdot (1/\alpha)^{\Omega(\log\log n)}$ total time.*

Proof. We instantiate our construction using the following parameters. The number of edges from a vertex to its ith child is $(1/\epsilon)^i$ ($p = 1/\epsilon$). Each vertex of H has $t = 1/2 \cdot \log_p n$ children. The tree H has depth $d = \frac{\log(101\alpha)}{2\log(1-\epsilon)} - 2 \geq 1$. Note that $d = \Theta(\log(1/\alpha))$. Finally, S_0, S_1 have $s = n/4$ leaves.

Let us now bound the size of the graph. The number of leaves of H is

$$t^d = (1/2 \cdot \log_{1/\epsilon} n)^d = \left(\frac{\log n}{2\log 1/\epsilon}\right)^{\Theta(\log(1/\alpha))} = \log^{\Theta(\log(1/\alpha))} n = (1/\alpha)^{\Theta(\log\log n)}, \qquad (2)$$

where in the third step we use the fact that $\frac{\log n}{2\log 1/\epsilon} = \log^{\Theta(1)} n$ for sufficiently large n.

R, S_0 and S_1 have $n/4$ vertices each. By Equation (2) and the assumption on α, H has $o(n)$ vertices, and so with the additional isolated vertices, the graph has exactly n vertices.

The number of edges incident to R, S_0 and S_1 is $O(n)$. The number of children of each internal vertex of H is

$$\Theta(p^t) = \Theta(p^{1/2 \cdot \log_p n}) = \Theta(n^{1/2}).$$

Thus, the total number of edges in H is $(1/\alpha)^{\Theta(\log\log n)} \cdot n^{1/2} = n^{o(1)} \cdot \Theta(n^{1/2})$. Hence, we conclude that the graph has n vertices and $O(n)$ edges.

Observe that as we add edges, leaves $\ell_1, \ell_2, \ldots$ become reachable from R exactly in the order of their indices. Fix any leaf ℓ_j of H. Denote by π^b and π^a, respectively, the PageRank vectors just before ℓ_j is reachable from R and just after all edges on the path from R to ℓ_j are added.

We use the interpretation of PageRank based on probability mass. Before ℓ_j is reachable from R, it may receive probability mass only from its ancestors in H. Hence,

$$\pi^b_{\ell_j} \le (d+1)/n = \Theta(\log(1/\alpha))/n = o(\log n)/n.$$

Moreover, since PageRank is a ϵ fraction of the probability mass entering each vertex, by Lemma 7,

$$\pi^a_{\ell_j} \ge \epsilon \cdot (1-\epsilon)^{2d+2}/4 = \epsilon \cdot (1-\epsilon)^{\frac{\log(101\alpha)}{\log(1-\epsilon)}-2}/4 = 101 \cdot \epsilon \cdot \alpha \cdot (1-\epsilon)^{-2}.$$

The increase to PageRank of ℓ_j is thus at least $\pi^a_{\ell_j} - \pi^b_{\ell_j} \ge 100 \cdot \epsilon \cdot \alpha(1-\epsilon)^{-2}$. Hence, after the insertion there is at least $100\alpha(1-\epsilon)^{-2}/4$ "new" probability mass at ℓ_j. Since every two hop path from j leads to a leaf in $S_{j \bmod 2}$, each of these leaves will receive a least

$$100 \cdot \alpha \cdot (1-\epsilon)^{-2}/4 \cdot (1-\epsilon)^2/s = 100 \cdot \alpha/(4s).$$

new probability mass (since only $(1-\epsilon)$ fraction of the probability mass is transferred along each hop). By Observation 6 all of that probability mass ends up increasing the PageRank of the leaf. Therefore the PageRank of each of these s leaves increases by $100 \cdot \alpha/(4s) = 100 \cdot \alpha/(4 \cdot n/4) = 100 \cdot \alpha/n$.

We now use Lemma 8 with $v_1 = \pi^b$, $v_2 = \pi^a$ and $\tilde{v}_1$ and $\tilde{v}_2$ being any PageRank vectors giving α-additive approximation and infer that $\Omega(n)$ coordinates of any approximate PageRank vector must be updated in order to maintain α-additive approximation. This happens for each leaf of H, and so by Equation (2) the Lemma follows. ◀

3.2 Lower Bound for Maintaining Multiplicative Approximation

▶ **Theorem 3.** *There exists a sequence of $\Theta(n)$ edge insertions applied to an initially empty graph on n vertices for which the following holds. For any constant $\delta > 0$, any algorithm that maintains a vector $\tilde{\pi} \in \mathbb{R}^n$ such that $(1/2)\pi_v < \tilde{\pi}_v \le 2\pi_v$ at all time steps, must take time $\Omega(n^{2-\delta})$ to process the sequence. In particular, the amortized update time of any such algorithm is $\Omega(n^{1-\delta})$.*

Proof. We instantiate our construction as follows. Each non-leaf vertex v of H has exactly $t = \delta/2 \log n/\log\log n$ children, with $(\log^2 n)^i$ parallel directed edges from v to the i-th child of v ($p = \log^2 n$). It follows that the total outdegree of each internal vertex in H is $O(n^\delta)$. The depth of H is set to be $d = \log_t(n^{1-2\delta}) = \Theta(\log n/\log\log n)$, so that H has $n^{1-2\delta}$ vertices, and the total number of edges in H is $O(n^{1-\delta})$. Finally, both S_0 and S_1 have $s = n^{1-2\delta}$ vertices.

Fix a leaf ℓ_j of H and consider the state of the algorithm right after all on the path from the root of H to ℓ_j have been added. By Lemma 7, the probability mass entering ℓ_j is at least.

$$(1-\epsilon)^{2d+2}/4 = (1-\epsilon)^{\Theta(\log n/\log\log n)} = n^{\Theta(-1/\log\log n)}.$$

Out of this probability mass a constant fraction reaches the leaves of $S_{j \bmod 2}$. In particular, the PageRank of each such leaf is at least $\epsilon \cdot n^{\Theta(-1/\log\log n)}/n^{1-2\delta} \geq n^{-\delta}$.

Moreover, out of the probability mass from R the fraction that reaches ℓ_j is at least

$$(1-1/p)^d = (1-1/\log^2 n)^{\Theta(\log n/\log\log n)} \geq 1 - 1/\log n.$$

out of all probability mass that reaches the leaves of H from R. Observe that compared to this probability mass (which is a constant), the total probability mass generated by all vertices of H is negligible. As a result, the ratio of probability mass that reaches $S_{j \bmod 2}$ to the probability mass that reaches $S_{(j+1) \bmod 2}$ is

$$\frac{1-1/\log n}{1/\log n} = \Theta(\log n).$$

This implies that when we add all edges on a path from R to ℓ_j, the PageRanks of leaves of $S_{j \bmod 2}$ increase by a factor of $\Theta(\log n)$ and so the PageRank estimates of all these $\Omega(n^{1-2\delta})$ vertices must be changed. Since a total of $m = O(n)$ edges are added, and since this occurs once for each of the $\Omega(n^{1-\delta})$ leaf vertices in H, we obtain a total of $\Omega(n^{2-3\delta})$ PageRank estimate updates, which is the desired result after rescaling δ by a constant. ◀

3.3 Lower bound for the ForwardPush algorithm

▶ **Theorem 9.** *Consider running the ForwardPush [25] algorithm whose error parameter is set to ensure that the algorithm maintains additive α approximation of PageRank. For any $\delta > 0$, each sufficiently large $n \geq 1$ and $\epsilon \in (0.01, 0.99)$ there exists a graph on n vertices and a sequence of $O(n)$ edge insertions, such that the algorithm runs in $\Omega(n^{2-\delta})$ time.*

Proof. We use our construction with the same settings as in the proof of Theorem 3. Specifically, $t = \delta/2 \log n/\log\log n$, $p = \log^2 n$, $d = \log_t(n^{1-2\delta}) = \Theta(\log n/\log\log n)$ and $s = n^{1-2\delta}$.

The ForwardPush algorithm can be explained using the probability mass interpretation. The algorithm maintains a *residual* on each vertex u, denoted by R_u. This residual can be positive or negative. Initially, the residual of each vertex is $1/n$.

The residual is a probability mass that still has to be pushed to the neighbors of u. The algorithm maintains two invariants

1. $|R_u| \leq \gamma \deg(u)$ for each vertex $u \in V$, where γ is an accuracy parameter.
2. If we keep pushing the residuals, the PageRank estimates converge to the exact PageRank values.

For any vertex u that violates the invariant, that is satisfies $|R_u|/\deg(u) > \gamma$, the algorithm executes a *push* operation, which takes time $\Theta(\deg(u))$ and pushes a $1-\epsilon$ fraction of the residual to the outneighbors of u and uses a ϵ fraction of the residual to increase the PageRank of u. The residual of u is then set to 0. Upon an insertion of an edge uv, the algorithm decreases R_u by $\Delta = \Theta(\pi_u)/\deg(u)$ and increases R_v by Δ. Then, it restores the invariant by executing push operations.

In the following part of the proof we use the following observation, which follows from the second algorithm invariant.

▶ **Observation 10.** *Fix a vertex v and denote by D_v the set of vertices that have a directed path to v. We assume $v \in D_v$. Then, the total additive error of the PageRank estimate maintained by the ForwardPush algorithm is at most $\sum_{u \in D_v} |R_u|$.*

By using the second algorithm invariant, we get that ForwardPush ensures that the total additive error is $\sum_{u \in V} |R_u| \leq \sum_{u \in V} \gamma \deg(u) = \gamma m$. Therefore, to ensure an additive α approximation of PageRank, we set $\gamma m \leq \alpha$, implying $\gamma \leq \alpha/m$. We note that it is easy to come up with an example where this analysis is tight up to a constant factor.

We now analyze ForwardPush algorithm on our hard instance. Since the number of edges in our graph is $\Theta(n)$, we invoke ForwardPush with the approximation parameter $\gamma = \Theta(1/n)$. We claim that with this value of γ, the residual values are propagated often enough so that over $\Theta(n)$ edge insertions described above, ForwardPush makes $\Omega(n^{2-\delta})$ updates.

We use the observations from the proof of Theorem 3 that the PageRank of a vertex c_i ($i \in \{0, 1\}$) is $n^{\Theta(-1/\log\log n)}$ and, as we add edges, increases by a $\Theta(\log n)$ factor each time we fully add a path from R to a leaf ℓ_j, such that $i = j \bmod 2$.

We now use Observation 10 to show that the ForwardPush maintains a constant factor approximate of the PageRank estimates of c_0 and c_1. Indeed, these vertices can only be reached from R, H or from themselves. We now bound the residuals of these vertices. The residuals of the vertices of R are set to 0 the moment each of these vertices performs the first push operation and are then never updated. The residual of each vertex v of H satisfies $|R_v|/\deg(v) \leq \alpha/m$ which implies $|R_v| \leq \Theta(\deg(v))/m = \Theta(n^{\delta-1})$. Finally, the residual of c_0 (and, similarly c_1) satisfies $|R_{c_0}|/\deg(c_0) \leq \alpha/m$, which gives $|R_{c_0}| \leq \Theta(n^{1-2\delta})/m = \Theta(n^{-2\delta})$. By applying Observation 10 we have that the additive error the PageRank estimates of c_0 and c_1 is at most

$$\Theta(n^{\delta-1}) \cdot \Theta(n^{1-2\delta}) + \Theta(n^{-2\delta}) = \Theta(n^{-\delta}).$$

These additive errors are negligible comared to the PageRanks of these vertices, which is $n^{\Theta(-1/\log\log n)}$. Hence, the algorithm maintains constant-factor estimates of the PageRanks of c_0 and c_1. As a result, when the exact PageRank values change by a factor of $\Theta(\log n)$, the algorithm updates their estimates. However, the ForwardPush algorithm only updates a PageRank estimate of a vertex u when either it executes a push operation on u or adds an outgoing edge from u. Since all outgoing edges of c_0 and c_1 have been added in the beginning, we get that the algorithm executes a push operation on c_0 for half of leaves of H. Each such operation takes $\Theta(\deg(c_0)) = \Theta(n^{1-2\delta})$ time and so the overall running time of the algorithm is $\Theta(n^{1-2\delta} \cdot n^{1-2\delta}) = \Theta(n^{2-4\delta})$ which, after tweaking δ by a constant factor, gives the desired. ◀

4 Approximating PageRank by Maintaining Dynamic Random Walks

In this section we review the algorithm for approximating PageRank by maintaining random walks. This algorithm is a dynamic version of Algorithm 1 and has been previously described by Bahmani et al. [4]. We provide a detailed proof of correctness of the algorithm, which to the best of our knowledge has not been included in any prior work.

The algorithm relies on maintaining $O_\epsilon(n \log n)$ random walks and re-sampling their parts as necessary. In this section, we present data structures that we use to efficiently maintain and re-sample those random walks. Section 4.1 presents our approach on an edge insertion, while Section 4.2 describes how our algorithms handle edge deletions. We being by describing the problem setup.

Setup. Following Proposition 5, to approximate the PageRank it suffices to sample $R = O(\log n/(\epsilon\alpha^2))$ PageRank walks from each vertex. A PageRank walk is a random walk w, whose length ℓ_w is sampled from geometric distribution with parameter $1 - \epsilon$. Even though a given walk may get re-routed after edge insertions or deletions, it is crucial that the the length of each walk remains **fixed** throughout the entire execution of the algorithm. Otherwise, it is easy to construct examples where the lengths of the maintained walks no longer follow the right distribution.

We maintain two types of data structures. For each each vertex v and $t = 0 \ldots O(\log n/\epsilon)$, we maintain a binary search tree $S_{v,t}$ which stores all the walks whose t-th vertex if v. For each edge e, we maintain the binary search tree W_e consisting of the walks passing through e.

4.1 Edge Insertion

When an edge (u, v) is inserted, we re-sample some of the walks passing through u. This re-sampling is done by first performing rejection sampling on each walk and, second, by choosing an appropriate position where each of the rejected walks should be re-sampled. Choosing an appropriate position from where to re-sample w is trivial in case when w passes through u once. However, it might be the case that w passes through that vertex multiple times, and a more careful consideration is required. At a high level, we iterate through all segments of w and for each segment of w that leaves u we toss a coin. Then, with probability $1/d_u$, where d_u is the degree of u after the update, we reroute w starting from the considered segment, and terminate the update procedure for w.

Each walk has a unique ID associated with it. These IDs are integers ranging from 1 through the number of walks we maintain. Each vertex and each edge keeps track of which walks are passing through them.

Given a vertex v and integers i and t, it will be convenient to be able to sample the i-th walk whose t-th vertex is v. It will become clear why such operation is needed when we describe how to handle edge insertions. To be able to implement this operation efficiently, we store the IDs of walks whose t-th vertex is v in a binary tree; we use $S_{v,t}$ to refer to this binary tree. Then, the i-th walk can be easily fetched via a search within that tree. The maximum value of t to consider is upper-bounded by the maximum length of the walks.

Assume that we insert an edge $e = (u, v)$. Let d_u be the out-degree of u *after* adding e. Consider a walk w that at some point got to u and continued to u's neighbors. If e was present in the graph at that point, with probability $1/d_u$ the walk w would have continued along e, and with probability $1 - 1/d_u$ the walk w would have chosen some other neighbor of u. However, w was sampled before e was in the graph, and our aim now is to correct this distribution and account for the insertion of e. The idea is to use rejection sampling, which we provide as Algorithm 2.

The for-loop on Line 3 of Algorithm 2 is in an efficient way of selecting walks passing through u and v that need to be re-sampled. Since the length of each walk follows a geometirc distribution with parameter $1 - \epsilon$, it is easy to see that with high probability the walks have length $O(\log n/\epsilon)$, and hence $\ell \in O(\log n/\epsilon)$.

Remark: To the best of our understanding, on an insertion of an edge (u, v), the prior work [24] re-samples a walk passing through u from the first occurrence of u in the walk, if there is any such occurrence (for details, see [24]). Such re-sampling does not account for the case when a walk passes through u multiple times and leads to biases in randomness.

4.2 Edge Deletions

Algorithm 3 presents our procedure executed after deleting an edge.

Let e be a deleted edge, and let $W_e \subseteq W$ be the list of walks passing through e. Clearly each $w \in W_e$ needs to be rerouted. The following lemma states that W updated by executing Algorithm 3 is a set of independent random walks.

Algorithm 2 A procedure executed after edge $e = (u, v)$ is inserted.

```
1: $W \leftarrow \emptyset$
2: Let $\ell$ be the length of longest generated walk.
3: for $t = 1 \dots \ell$ do
4:     Sample each walk from $S_{u,t}$ with probability $1/d_u$ in the following way. First,
       select an integer $r_{u,t}$ from the binomial distribution with parameters $|S_{u,t}|$ and
       $1/d_u$. Second, select $r_{u,t}$ integers uniformly at random and without repetition
       from $[1, |S_{u,t}|]$. Then, for each of those integers $i$ select the $i$-th walk from $S_{u,t}$.
       If $e$ is an undirected edge, apply the same steps for $S_{v,t}$.
5:     For each walk $w$ selected in the last step such that $w \notin W$, add $w$ to $W$ and
       label $w$ by $t$.
6: end for
7: for each $w \in W$ do
8:     Let $j$ be the label remembered for $w$ on Line 5.
9:     Generate walk $w'$ with the following properties:
         - The walks $w$ and $w'$ have the same length.
         - The vertex-prefixes of length $j$ of $w$ and $w'$ are the same.
         - After that prefix, if $w$ has more than $j$ vertices, $w'$ walks along $e$.
         - The remaining edges of $w'$ are chosen randomly, i.e., the rest of $w'$ is a newly
           generated random walk.
10:    Update the data structures by removing $w$ and inserting $w'$.
11: end for
```

Algorithm 3 A procedure executed after edge e is deleted.

```
Let $W_e \subseteq W$ be the list of walks passing through $e$.
for $w \in W_e$ do
    Let $w_p$ be the longest prefix of $w$ not containing $e$.
    Let $w'$ be a walk of length $|w|$ such that $w'$ has $w_p$ as its prefix, and the
    remainder of $w'$ is a random walk.
    To update $W$, remove $w$ from $W$ and the corresponding data structures, and
    insert $w'$.
end for
```

▶ **Lemma 11.** *Let W be the set of walks that our algorithm maintains. Assume that e gets deleted, and let W' be the updated list of walks as described in Algorithm 3. If W consists of random walks sampled independently, then W' is also a set of random walks sampled independently.*

Proof. The edges of walks throughout the algorithm are sampled independently of each other, so walks are independent by construction. We focus on showing how deletion of an edge affects randomness of a single walk.

Consider a walk $w \in W$ originating at vertex w_1. Let w_i be the i-th vertex of w, $w_{1\dots i}$ be the prefix of length i of w, and k be the length of w. Walk w is random iff for each $i \geq 2$ and each $u \in N(w_{i-1})$ it holds

$$\Pr\left[w_i = u \mid w_{1\dots i-1}\right] = \frac{1}{d(w_{i-1})}. \tag{3}$$

Let w' be the updated walk w, $d'(v)$ be the updated degree of vertices after e gets deleted and u' be a neighbor of w'_{i-1} after deletion of e. Note: we are **not** assuming that w contains e, so it might be the case that $w = w'$. We want to show that $\Pr\left[w'_i = u' \mid w'_{1\ldots i-1}\right] = 1/d'(w'_{i-1})$. We have

$$\begin{aligned}
&\Pr\left[w'_i = u' \mid w'_{1\ldots i-1}\right] && (4)\\
=&\Pr\left[w'_i = u' \mid w'_{1\ldots i-1}, e \in w_{1\ldots i}\right] \cdot \Pr\left[e \in w_{1\ldots i}\right] && (5)\\
&+\Pr\left[w'_i = u' \mid w'_{1\ldots i-1}, e \notin w_{1\ldots i}\right] \cdot \Pr\left[e \notin w_{1\ldots i}\right]. && (6)
\end{aligned}$$

Analyzing (5). We first handle (5). Recall that w' is constructed by keeping only the prefix of w up to the first occurrence of e, and the rest of the walk of w' is random and independent of any other state of the algorithm (see Algorithm 3). Hence, we have

$$\Pr\left[w'_i = u' \mid w'_{1\ldots i-1}, e \in w_{1\ldots i}\right] = \frac{1}{d'(w'_{i-1})}.$$

Analyzing (6). Now consider term (6). If $w_{1\ldots i}$ does not contain e, then $w'_{1\ldots i} = w_{1\ldots i}$ and we have

$$\begin{aligned}
&\Pr\left[w'_i = u' \mid w'_{1\ldots i-1}, e \notin w_{1\ldots i}\right]\\
=&\Pr\left[w_i = u' \mid w_{1\ldots i-1}, e \notin w_{1\ldots i-1}, e \neq \{w_{i-1}, w_i\}\right].
\end{aligned}$$

There are two cases:

(a) Case $w_{i-1} \notin e$: from (3) we have

$$\begin{aligned}
&\Pr\left[w_i = u' \mid w_{1\ldots i-1}, e \notin w_{1\ldots i-1}, e \neq \{w_{i-1}, w_i\}, w_{i-1} \notin e\right]\\
=&\Pr\left[w_i = u' \mid w_{1\ldots i-1}, e \neq \{w_{i-1}, w_i\}, w_{i-1} \notin e\right]\\
=&\frac{1}{d(w_{i-1})} = \frac{1}{d'(w_{i-1})} = \frac{1}{d'(w'_{i-1})}.
\end{aligned}$$

In the last chain of equalities we used that once we condition on $w_{1\ldots i-1}$, then (3) is a function of only w_{i-1} and not on any other content of $w_{1\ldots i-1}$, e.g., whether $e \in w_{1\ldots i-1}$ or not.

Note: The choice of e is independent of our data structures and the randomness the algorithm uses. However, in the case of non-oblivious adversary, i.e., in case of the adversary who sees the state of our algorithm, the updated edge e could be chosen based on the randomness used to generate w, and hence the above sequence of equalities would not hold.

(b) Case $w_{i-1} \in e$: we have the following

$$\begin{aligned}
&\Pr\left[w_i = u' \mid w_{1\ldots i-1}, e \notin w_{1\ldots i-1}, e \neq \{w_{i-1}, w_i\}, w_{i-1} \in e\right]\\
=&\frac{\Pr\left[w_i = u' \wedge e \neq \{w_{i-1}, w_i\} \mid w_{1\ldots i-1}, e \notin w_{1\ldots i-1}, w_{i-1} \in e\right]}{\Pr\left[e \neq \{w_{i-1}, w_i\} \mid w_{1\ldots i-1}, e \notin w_{1\ldots i-1}, w_{i-1} \in e\right]}\\
=&\frac{1/d(w_{i-1})}{(d(w_{i-1})-1)/d(w_{i-1})} = \frac{1}{d(w_{i-1})-1} = \frac{1}{d'(w'_{i-1})}.
\end{aligned}$$

Showing (3) **for** w'**.** The analysis of (5) and (6) together with (4) implies

$$\begin{aligned}&\Pr\left[w'_i = u' \mid w'_{1\ldots i-1}\right]\\ =&\frac{1}{d'(w'_{i-1})}\cdot \Pr\left[e \in w_{1\ldots i}\right] + \frac{1}{d'(w'_{i-1})}\cdot \Pr\left[e \notin w_{1\ldots i}\right]\\ =&\frac{1}{d'(w'_{i-1})}.\end{aligned}$$

◀

4.2.1 Re-sampling Walks from Scratch

We now give a simple example that shows why re-sampling affected walks from scratch after a deletion would not properly maintain random walks. We note that this approach was suggested as a valid alternative by Bahmani et al. [4].

Consider a path graph on 5 vertices; let the graph be $1-2-3-4-5$. Consider a random walk w of length 2 originating at vertex 3 and visiting vertices w_1, w_2, w_3, i.e., $w_1 = 3$. Next, a deletion of $e = \{4,5\}$ occurs. Let w' be obtained from w as follows: if w contains e, then w' is a new random walk of length 2 originating at 3; otherwise, w' equals w. Now, if we denote the vertices on w' by w'_1, w'_2, w'_3, we have

$$\begin{aligned}\Pr\left[w'_2 = 4\right] =& \Pr\left[w'_2 = 4 \mid \{4,5\} \notin w\right]\Pr\left[\{4,5\} \notin w\right]\\ &+ \Pr\left[w'_2 = 4 \mid \{4,5\} \in w\right]\Pr\left[\{4,5\} \in w\right]\\ =&\Pr\left[w_2 = 4 \mid \{4,5\} \notin w\right]\Pr\left[\{4,5\} \notin w\right]\\ &+ \Pr\left[w'_2 = 4 \mid \{4,5\} \in w\right]\Pr\left[\{4,5\} \in w\right]\\ =&\Pr\left[w_2 = 4 \text{ and } \{4,5\} \notin w\right] + \frac{1}{2}\cdot\frac{1}{4}\\ =&\frac{1}{4}+\frac{1}{8}.\end{aligned}$$

However, for w' to be random it should hold $\Pr\left[w'_2 = 4\right] = 1/2$.

5 Near-Optimal Additive Approximation Algorithm

In this section, we analyze the algorithm from Section 4 in the context of dynamically maintaining *additive* approximation of PageRank. Namely, we show that when considering the incremental or decremental setting for directed graphs, an α additive PageRank approximation can be maintained in $(1/\alpha)^{O_\epsilon(\log\log n)}$ amortized update time, even for an adversarially chosen graph and a sequence of edge updates. Perhaps surprisingly, Theorem 1 shows that, for a constant ϵ, this running time complexity is essentially tight.

▶ **Theorem 2.** *For any $\epsilon \in (0,1)$, there is an algorithm that with high probability explicitly maintains an α additive approximation of PageRank of any graph G in either incremental or decremental setting. The algorithm processes the entire sequence of updates in $O(m) + n \cdot (1/\alpha)^{O_\epsilon(\log\log n)}$ total time and works correctly against an oblivious adversary.*

Our new analysis is based on two ideas. First, we show that if we *limit the lengths* of walks in Algorithm 1 to a constant, we obtain a constant additive approximation of the PageRank vector. This is thanks to the fact that a constant fraction of all walks have length $O(1/\epsilon)$, and so this truncation only affects a constant factor of the walks.

▶ **Lemma 12.** *Let π be the PageRank of a directed graph G. Then, with high probability, Algorithm 1 for $\ell = \lceil 2/\epsilon \cdot \log(2/(\alpha\epsilon))\rceil$ outputs a vector π_{ADD} such that $\|\pi - \pi_{ADD}\|_1 \le 5\frac{\alpha}{1-\epsilon}$.*

To keep the flow of high-level ideas uninterrupted, the proof of Lemma 12 is given in Section 5.1.

The second idea is an observation which bounds the maximum number of times a walk can be affected by adding edges (edge deletions can use a symmetric argument). To explain the idea let us see what happens when we want to maintain a random outgoing edge e of a vertex undergoing insertions of outgoing edges. Clearly when we insert the d-th outgoing edge we need to update e to be equal to d with probability $1/d$. By a harmonic sum argument, the expected number of times e needs to be updated in the course if k insertions is only $O(\log k)$. We generalize this argument to walks of length ℓ as follows.

▶ **Lemma 13.** *Let G be a directed graph undergoing edge insertions (or deletions). The total number of times a random walk of length ℓ is being regenerated is bounded by $O(\log^{\ell} n)$ in expectation.*

Proof. We are going to prove this bound by induction, i.e., let us denote by $f(i)$ the upper bound on expected number of times the walk of length i is regenerated. Consider a random walk w of length 1 starting in a vertex v. Consider insertion of an edge incident to v. The probability that w is regenerated at this moment is $1/d_v$. As we consider incremental setting the expected number of times w is regenerated is bounded by

$$f(1) = \sum_{i=1}^{n} \frac{1}{i} \leq \ln n.$$

Now consider a walk w of length ℓ starting at v. Similarly as above we can bound the number of changes to w as

$$f(\ell) = \sum_{i=1}^{n} \frac{1}{i} \cdot f(\ell-1) \leq \ln n \cdot f(\ell-1) = \ln^{\ell} n,$$

what finishes the proof. Symmetric argument can be applied in the decremental case. ◀

The above lemma implies that for $\ell = \lceil 2/\epsilon \cdot \log(2/(\alpha\epsilon)) \rceil$ the amortized cost of maintaining each walk is $(1/\alpha)^{O(\log\log n)}$ for a constant ϵ. As we generate $O(n \log n)$ walks in Algorithm 1 the total cost of maintaining $5\alpha/(1-\epsilon)$-approximation in incremental or decremental setting is $O(m + n \cdot (1/\alpha)^{O(\log\log n)})$.

5.1 Proof of Lemma 12

Define $\hat{\ell} = \lceil 2/\epsilon \cdot \log(2/(\alpha\epsilon)) \rceil$. Let $\tilde{\pi}$ be the output of Algorithm 1 for $\ell = \infty$, and π_{ADD} the output for $\ell = \hat{\ell}$. As discussed, it is known, e.g., see [4, 16], that $|\pi_v - \tilde{\pi}_v| \leq \alpha\pi_v$. As $\sum_v \pi_v = 1$, this further implies $\|\pi - \tilde{\pi}\|_1 \leq \alpha$.

Next, we compare π_{ADD} and $\tilde{\pi}$. Difference between these two vectors can be expressed by the following two quantities: (1) $|W|$, which in turn affects the scaling on Line 5; and (2) the value of $\mathbf{X}_v$, which affects the numerator on Line 5. We analyze both of these quantities.

Analysis for $|W|$. For $\ell = \hat{\ell}$, a walk has length at most $\hat{\ell}$ with probability $\epsilon \sum_{j=0}^{\hat{\ell}} (1-\epsilon)^j = 1 - (1-\epsilon)^{\hat{\ell}+1} \geq 1 - \epsilon/2$, where we used that $1 - x \leq e^{-x}$ for $x \in [0, 1/2]$. Hence, $\mathbb{E}[|W|] \geq nR(1-\epsilon/2)$. By using a Chernoff bound we can prove that with high probability it holds $|W| \geq nR(1-\epsilon)$. The proof proceeds as follows. In the summation above, there are only ℓ different values of j that affect $\mathbb{E}[|W|]$. For a fixed j, the contribution to $|W|$ can be expressed as a sum of **independent** 0/1 random variables – a random variable per each of the nR walks, denoting whether the given walk has length length j or not. Hence, for a fixed j we apply the Chernoff bound to show it concentrates well, and then by the union bound over all ℓ values of j we get the desired concentration for $|W|$.

Analysis for $\mathbf{X}_v$. By definition, π_{ADD} only accounts for the contribution to $\mathbf{X}_v$ by the appearances of v which are within walks of length at most ℓ; $\mathbf{X}_v$ is defined in Algorithm 1. Let $\mathbf{Y}_v$ be the appearances of v for which π_{ADD} does not but $\tilde{\pi}$ does account for.

Now, we upper-bound $\sum_v \mathbf{Y}_v$:

$$\begin{aligned}\mathbb{E}\left[\sum_v \mathbf{Y}_v\right] =& nR(1-\epsilon)^{\hat{\ell}+1}\cdot(\hat{\ell}+1) + \sum_{j=\hat{\ell}+2}^{\infty} nR(1-\epsilon)^j\\ \le& 2nR\alpha + nR(1-\epsilon)^{\hat{\ell}+2}\sum_{j=0}^{\infty}(1-\epsilon)^j\\ =& 2nR\alpha + \frac{nR}{\epsilon}(1-\epsilon)^{\ell+2}\\ \le& 2nR\alpha + nR\epsilon\alpha^2/4\\ \le& 3nR\alpha.\end{aligned}$$

In the derivation above, we used $(1-\epsilon)^{\hat{\ell}+1}(\hat{\ell}+1) \le (\alpha\epsilon/2)^2(\hat{\ell}+1) \le (\alpha\epsilon/2)^2 2\hat{\ell} \le 2\alpha$. To prove that $\sum_v \mathbf{Y}_v \le 4nR\alpha$ with high probability, it suffices to proceed the same way as for our analysis of $\mathbb{E}[|W|]$. In the analysis, we need the observation that $\sum_{j>c\log n/\epsilon} nR(1-\epsilon)^j < 1/n$ for a sufficiently large constant c. In other words, there are only $O(\log n)$ different values of j that substantially contribute to $\sum_v \mathbf{Y}_v$ and over which is needed to take the union bound.

Our analysis now implies that additive approximation of Algorithm 1 for $\ell = \hat{\ell}$ is with high probability upper-bounded by $\frac{\alpha}{1-\epsilon} + 4\frac{\alpha\epsilon}{1-\epsilon} \le 5\frac{\alpha}{1-\epsilon}$. The first term is coming from the fact that π_{ADD} is computed by rescaling $\mathbf{X}_v$ by $|W|/\epsilon \ge (1-\epsilon)nR/\epsilon$ as opposed to rescaling by nR/ϵ, as it is done when computing $\tilde{\pi}$. The second term is coming from the fact that the loss between $\tilde{\pi}$ and π_{ADD} in the numerator of Line 5 is at most $4nR\alpha$ with high probability, which is divided by $|W|/\epsilon \ge nR(1-\epsilon)/\epsilon$.

6 Efficient Multiplicative Approximation in Undirected Graphs

In this section, we describe how to maintain approximate PageRank of undirected graphs under edge deletions and insertions even if the goal is to maintain a multiplicative approximation. Our approach takes polylog n time per update and is also based on the algorithm from Section 4.

▶ **Theorem 4.** *For any $\epsilon \in (0,1)$, there is an algorithm that with high probability explicitly maintains a $1+\alpha$ multiplicative approximation of PageRank of any undirected graph G in the fully dynamic setting. The algorithm handles each update in $O(\log^5 n/(\epsilon^2\alpha^2))$ time and works correctly against an oblivious adversary.*

Our analysis relies on the following (folklore) claim, which states that the number of the walks passing through an edge is fairly small.

▶ **Lemma 14** (Folklore). *Let G be an undirected graph. Consider a set of random walks W of length $\ell < n$ each, such that there are d_v walks originating at vertex v. Then, with high probability an edge e is contained in $O(\ell \cdot \log n)$ of those walks.*

Proof. Observe that the number of walks in W originating at each vertex v is proportional to the stationary distribution of v. Hence, the number of walks of W whose i-th vertex is v in expectation equals d_v, for each $1 \leq i \leq \ell$. Therefore, the number of walks of W whose i-th edge is $e = \{u, v\}$ (either as $u \to v$ or $v \to v$) in expectation equals 2, for each $1 \leq i \leq \ell$.

Let $X_{e,i}$ be the number of walks whose i-th edge equals i. From our discussion, $\mathbb{E}[X_{e,i}] = 2$. Also, $X_{e,i}$ is a sum of 0/1 independent random variables $Y_{v,j,i}$, where $Y_{v,j,i}$ means that the i-th edge of the j-th walk originating at v equals e. Hence, by applying the Chernoff bound, we obtain that with high probability it holds that $X_{e,i} \in O(\log n)$. By taking the union bound over all $1 \leq i \leq \ell$ and over all the vertices, we prove the desired claim. ◀

As a direct consequence of Lemma 14 we obtain the following claim.

▶ **Corollary 15.** *Consider $n \cdot t$ independent random walks of length $\ell \in O(\log n/\epsilon)$ such that from each vertex there are t walks originating. Then, with high probability an edge e is contained in $O(t \log^2 n/\epsilon)$ of those walks.*

In Section 4, we describe how to update our data structures in $O(\ell \cdot \log n)$ time per an update of an ℓ-length walk. Since Algorithm 1 runs $t = R = O(\log n/(\epsilon\alpha^2))$ random walks per vertex, by Corollary 15 there are $O(\log^3 n/(\epsilon\alpha^2))$ walks passing through each edge. Thus by the fact that walks have lengths $O(\log n/\epsilon)$ with high probability, the dynamic algorithm requires $O(\log^5 n/(\epsilon^2\alpha^2))$ time for each update, which yields Theorem 4.

References

1 Madhav Aggarwal, Bingyi Zhang, and Viktor Prasanna. Performance of local push algorithms for personalized pagerank on multi-core platforms. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 370–375. IEEE, 2021.

2 Reid Andersen, Christian Borgs, Jennifer Chayes, John Hopcraft, Vahab S Mirrokni, and Shang-Hua Teng. Local computation of PageRank contributions. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 150–165. Springer, 2007.

3 Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using pagerank vectors. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 475–486. IEEE, 2006.

4 Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized PageRank. *Proc. VLDB Endow.*, 4(3):173–184, December 2010. doi:10.14778/1929861.1929864.

5 Bahman Bahmani, Ravi Kumar, Mohammad Mahdian, and Eli Upfal. PageRank on an evolving graph. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 24–32, 2012.

6 Marco Bressan, Enoch Peserico, and Luca Pretto. Sublinear algorithms for local graph centrality estimation. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 709–718. IEEE, 2018.

7 Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998. doi:10.1016/S0169-7552(98)00110-X.

8 Soumen Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *Proceedings of the 16th international conference on World Wide Web*, pages 571–580, 2007.

9 Wentian Guo, Yuchen Li, Mo Sha, and Kian-Lee Tan. Parallel personalized PageRank on dynamic graphs. *Proceedings of the VLDB Endowment*, 11(1):93–106, 2017.

10 Kyung Soo Kim and Yong Suk Choi. Incremental iteration method for fast PageRank computation. In *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*, IMCOM '15, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2701126.2701165.

11 Qun Liao, ShuangShuang Jiang, Min Yu, Yulu Yang, and Tao Li. Monte Carlo based incremental PageRank on evolving graphs. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 356–367. Springer, 2017.

12 Peter Lofgren. On the complexity of the monte carlo method for incremental pagerank. *Inf. Process. Lett.*, 114(3):104–106, 2014. doi:10.1016/J.IPL.2013.11.006.

13 Peter Lofgren, Siddhartha Banerjee, and Ashish Goel. Personalized pagerank estimation and search: A bidirectional approach. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pages 163–172, 2016.

14 Peter Lofgren and Ashish Goel. Personalized PageRank to a target node. *arXiv preprint*, 2013. arXiv:1304.4658.

15 Peter A Lofgren, Siddhartha Banerjee, Ashish Goel, and C Seshadhri. FAST-PPR: Scaling personalized PageRank estimation for large graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1436–1445, 2014.

16 Jakub Łącki, Slobodan Mitrović, Krzysztof Onak, and Piotr Sankowski. Walking randomly, massively, and efficiently. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 364–377, 2020.

17 Amit Pathak, Soumen Chakrabarti, and Manish Gupta. Index design for dynamic personalized pagerank. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1489–1491. IEEE, 2008.

18 Ryan A Rossi and David F Gleich. Dynamic pagerank using evolving teleportation. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 126–137. Springer, 2012.

19 Subhajit Sahu, Kishore Kothapalli, and Dip Sankar Banerjee. Dynamic batch parallel algorithms for updating pagerank. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1129–1138. IEEE, 2022.

20 Scott Sallinen, Juntong Luo, and Matei Ripeanu. Real-time pagerank on dynamic graphs. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 239–251, 2023.

21 Hanzhi Wang, Zhewei Wei, Junhao Gan, Sibo Wang, and Zengfeng Huang. Personalized PageRank to a target node, revisited. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 657–667, 2020.

22 Hanzhi Wang, Zhewei Wei, Ji-Rong Wen, and Mingji Yang. Revisiting local computation of pagerank: Simple and optimal. In *STOC'24*, 2024.

23 Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, January 2008. doi:10.1007/s10115-007-0114-2.

24 Zexing Zhan, Ruimin Hu, Xiyue Gao, and Nian Huai. Fast incremental pagerank on dynamic networks. In *International Conference on Web Engineering*, pages 154–168. Springer, 2019.

25 Hongyang Zhang, Peter Lofgren, and Ashish Goel. Approximate personalized PageRank on dynamic graphs. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1315–1324, 2016.