Approximate Personalized PageRank on Dynamic Graphs

Hongyang Zhang
Department of Computer Science
Stanford University
hongyang@cs.stanford.edu

Peter Lofgren
Department of Computer Science
Stanford University
plofgren@cs.stanford.edu

Ashish Goel
Department of Management Science and Engineering
Stanford University
ashishg@stanford.edu

ABSTRACT

We propose and analyze two algorithms for maintaining approximate Personalized PageRank (PPR) vectors on a dynamic graph, where edges are added or deleted. Our algorithms are natural dynamic versions of two known local variations of power iteration. One, Forward Push, propagates probability mass forwards along edges from a source node, while the other, Reverse Push, propagates local changes backwards along edges from a target. In both variations, we maintain an invariant between two vectors, and when an edge is updated, our algorithm first modifies the vectors to restore the invariant, then performs any needed local push operations to restore accuracy.

For Reverse Push, we prove that for an arbitrary directed graph in a random edge model, or for an arbitrary undirected graph, given a uniformly random target node t, the cost to maintain a PPR vector to t of additive error ε as k edges are updated is $O(k + \overline{d}/\varepsilon)$, where $\overline{d}$ is the average degree of the graph. This is $O(1)$ work per update, plus the cost of computing a reverse vector once on a static graph. For Forward Push, we show that on an arbitrary undirected graph, given a uniformly random start node s, the cost to maintain a PPR vector from s of degree-normalized error ε as k edges are updated is $O(k + 1/\varepsilon)$, which is again $O(1)$ per update plus the cost of computing a PPR vector once on a static graph.

Categories and Subject Descriptors

G.2.2 [**Mathematics of Computing**]: Discrete Mathematics—*Graph Algorithms*

Keywords

Personalized PageRank; Online algorithm; Local push.

1. INTRODUCTION

Personalized PageRank (PPR) models the relevance of nodes in a network from the point of view of a given node. It has applications in search [13, 12], friend recommendations [4, 11], community detection [22, 2], video recommendations [6], and other applications. Because PPR is expensive to compute at query time, several authors have proposed precomputing it for each user and storing it [13, 7, 8]. However, in practice graphs are dynamic, for example on a social network users are constantly adding new edges to the network, so we need a method of updating pre-computed PPR values. In this work, we propose two new algorithms for updating pre-computed PPR vectors and give the first rigorous analysis of their running time.

There are four main algorithms for computing PPR [15], of which only one has an analysis for dynamic graphs prior to our work. The first is power iteration [21], but it is very slow, requiring $\Omega(m)$ time per user (where m is the number of edges), so it can't be used efficiently for PPR even on static graphs. The second, Forward Push [7, 2], is a local variation of power iteration which starts from the source user (the node whose point of view we take) and pushes probability mass forwards along edges. Berkin [7] proposed precomputing Forward Push vectors from many source nodes, and Ohsaka et al. [20] proposed an algorithm for updating it on dynamic graphs, however no past work has analyzed the running time for maintaining Forward Push. The third, Reverse Push [13, 2], is an alternative variation of power iteration which starts at each target node and pushes values backwards along edges to improve estimates. Jeh and Widom [13] and Lofgren et al. [17] propose pre-computing Reverse Push vectors (or a variation on them), to enable efficient search, but no past work has proposed an algorithm for updating Reverse Push vectors on dynamic graphs. Finally, a fourth method of computing PPR is Monte-Carlo [3, 9], and for that algorithm Bahmani et al. [5] give an efficient algorithm for updating it on dynamic graphs, with running time analysis in a random edge arrival order model. In experiments, Ohsaka et al. [20] find that when maintaining very accurate PPR vectors, Monte-Carlo is slower than Forward-Push, motivating the analysis of updating Forward Push.

Our contribution is the first running time guarantees for updating Forward Push on undirected graphs and for updating Reverse Push on directed as well as undireted graphs. The analysis is challenging because a new edge at one node can cause that node to push, which can cause a cascade of other nodes to push. Understanding and bounding the size of this cascade required a novel amortized analysis. We allow for a worst case graph, but we assume that edge updates arrive in a random order to better capture performance in practice. The same assumption was used in the analysis of Monte Carlo [5] where the bounds it leads to were found

KDD '16, August 13-17, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-4232-2/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2939672.2939804

to model the running time in practice on Twitter. Alternatively, for undirected graphs we prove a worst-case amortized running time. In addition, for undirected graphs, we strengthen the analysis of Monte-Carlo on dynamic graphs [5] to a worst-case edge arrival order.

Our algorithms are simple to implement, and we present two experiments for forward push. One insight from our theoretical analysis is a different way of updating Forward Push values than the previously proposed method [20], and we find that our variation is 1.5-3.5 times faster than that method without sacrificing accuracy. In the second experiment, we evaluate forward push for the problem of finding the top K highest personalized PageRank from a source node. We compare to the Monte-Carlo method of Bahmani et al. [5] and found that the forward push is 4.5-12 times more efficient in storage compared to Monte-Carlo, and is able to do edge update 1.5 - 2.5 times faster.

The main idea of our algorithms is that both Forward Push and Reverse Push maintain an invariant between a pair of vectors, and when an edge is added or deleted, we first restore the invariant with an efficient change to the vectors, then perform local push operations as needed to control the error. One limitation of our analysis is that it applies to the pure versions of Forward Push and Reverse Push, while some prior work [13, 7] proposes combining Forward or Reverse push vectors together, but we believe our analysis would extend to that case. It would also be interesting to extend our analysis adapt the Personalized PageRank search index of [17] to dynamic graphs.

Results.

For the Reverse Push algorithm, we show that on a worst-case graph, for a uniform random target, the expected time required to maintain PPR estimates at accuracy ε to that target as k edge updates arrive in a random order is $O(k + k/(n\varepsilon) + \overline{d}/\varepsilon)$. Here ε is the desired additive error of the estimates, and $\overline{d}$ is the average degree of nodes in the graph. Since a random PPR value is $1/n$, values smaller than $1/n$ are not very meaningful. Hence typically $\varepsilon = \Omega(1/n)$ and our running time is $O(k + \overline{d}/\varepsilon)$. Since $\overline{d}/\varepsilon$ is the expected time required to compute Reverse Push from scratch [1, 15], we see that our incremental algorithm can maintain estimates after every edge update using $O(1)$ time per update and the time required to compute a single Reverse Push vector.

For the Forward Push algorithm, we show that on an arbitrary undirected graph and arbitrary edge arrival order, for a uniform random source node s, the worst-case running time to maintain a PPR vector from that source node is $O(k + k/(n\varepsilon) + 1/\varepsilon)$. Here ε is a bound on the degree-normalized error (which we define later). Typically $\varepsilon = \Omega(1/n)$, so this is $O(k + 1/\varepsilon)$. The cost of computing such a PPR vector from scratch is $O(1/\varepsilon)$ [2] so we again see that the cost to maintain the PPR vector over k updates is $O(1)$ per update plus the cost of computing it once from scratch.

One observation that follows from our work is that for Monte-Carlo, for an arbitrary undirected graph and arbitrary edge arrival order, the time required to maintain r random walks from every source node over k arrivals is $O(rk)$. This contrasts with a worst-case example constructed on directed graphs where the total running time to maintain these walks grows super-linear in k [14]. Our observation suggests a weaker yet still meaningful bound without the random edge assumption made in Bahmani et al. [5] for Monte-Carlo methods on undirected graphs.

2. PRELIMINARIES

Let $G = (V, E)$ be an unweighted directed graph. Let A denote the adjacency matrix of G and let D denote the diagonal matrix representing the outdegrees of all vertices in G. For a vertex v, let $N^{\text{OUT}}(v)$ denote the set of out neighbors of v and let $N^{\text{IN}}(v)$ denote the set of in neighbors of v. The personalized PageRank vector $\vec{\pi}_s$ for a source node s is defined as the unique solution of the following linear system ([21]):

$$\vec{\pi}_s = \alpha \cdot \vec{e}_s + (1-\alpha) \cdot A^{\intercal} D^{-1} \vec{\pi}_s \tag{1}$$

where α (a.k.a. the teleport probability) is a constant between 0 and 1, and $\vec{e_s}$ is the indicator vector with a single nonzero entry of 1 at s. For a pair of vertices s and t on G, we will use $\pi(s,t)$ to denote the personalized PageRank from s to t. This linear algebraic definition of personalized PageRank is equivalent to simulating a random walk. Start from the source node s, with probability $(1-\alpha)$, go to a uniformly chosen neighbor of the current node, or with probability α stop at the current node: $\pi(s,t)$ is the probability that a random walk from s stops at t [21]. In the above definitions, we assumed that the outdegree of every vertex is at least one for simplicity. If a vertex has no out neighbors, then the random walk transitions to s with probability $(1-\alpha)$ (cf. Gleich [10] for other ways to handle dangling nodes).

2.1 The edge arrival model

In the dynamic edge arrival model, we start with an initial graph and there is a sequence of edge updates one by one. Let $G_0 = (V_0, E_0)$ denote the initial graph. Let k denote the number of edge updates. When the i-th edge $e_i = (u_i, v_i)$ arrives, if e_i is already in G_{i-1}, then it will be deleted; else it will be added to G_{i-1}. Let $G_i = (V_i, E_i)$ denote the updated graph. Note that V_i can differ from V_{i-1} by at most two vertices. Let $d_i^{\text{OUT}}(u)$ denote the outdegree of $u \in V$ on G_i and $d_i^{\text{IN}}(u)$ denote its indegree. Also let D_i denote the diagonal matrix of the outdegrees of V_i. Let $\pi_i(s,t)$ denote the personalized PageRank from s to t on G_i. Let $n = |V|_0$ denote the number of vertices and $m = |E_0|$ denote the number of edges in the initial graph G_0. We will analyze the following two edge arrival models:

Random edge permutations of directed graphs [5].

To introduce this graph model in our notation, consider a uniformly random edge permutation of E_k. Then G_0 is the subgraph consisting of the first m edges in the permutation. And e_i is defined as the $(m+i)$-th edge in this permutation, for $i = 1, \ldots, k$. In a uniformly random edge stream, the personalized PageRank vector does not change by "too much" after an edge update [5]. This intuition is not true in the worst case — there exists a sequence of edge insertions such that for a single source node, the aggregate l_1 difference between the personalized PageRank vector before and after every edge insertion grows superlinear in the number of of nodes [14].

Arbitrary edge updates of undirected graphs.

For undirected graph, we observe a few interesting properties for personalized PageRank, which will be used later.

The first one exploits the fact that on an undirected graph, a random walk can be in either direction.

Proposition 1 (cf. Lemma 1 in Lofgren et al. [16]). *Let G be an undirected graph. Let s and t be two vertices of G. Then $\pi(s,t) \times d(s) = \pi(t,s) \times d(t)$.*

Here we dropped the superscript on d since there is no direction for undireced edges.

Proposition 2. *Let $G = (V, E)$ be an undirected graph and let t be a vertex of V, then $\sum_{x \in V} \pi(x,t)/d(t) \leq 1$.*

Proof. Note that,

$$\sum_{x \in V} \frac{\pi(x,t)}{d(t)} = \sum_{x \in V} \frac{\pi(t,x)}{d(x)} \leq \sum_{x \in V} \pi(t,x) = 1.$$

where we used Proposition 1. □

2.2 Local push algorithms

The random walk interpretation of personalized PageRank leads to the following general class of local computation algorithms: start with all the probability mass on the source node of the graph, and iteratively push this mass out to neighboring nodes, getting progressively better approximations. In these "local push" algorithms, we typically maintain a "residual" at every node, which is mass that has been received but not yet been pushed out from that node, as well as an "estimate", which is mass that has been both received and pushed out (after accounting for the teleport probability and the outdegrees), we will now precisely describe how this local approach can be used in the forward and reverse direction.

2.2.1 Forward push

Given a source node s, the forward push algorithm maintains an estimate $\vec{P}(s,t)$ of $\pi(s,t)$ for each target $t \in V$. It also maintains a residual $\vec{R}(s,t)$ for t. Let $\vec{P}(s) = \vec{P}(s,\cdot)$ denote the the vector of estimates and $\vec{R}(s)$ the vector of residuals. The estimates and residuals satisfy the following invariant property (cf. Section 3 in Lofgren [2]):

$$\pi(s,t) = \vec{P}(s,t) + \sum_{x \in V} \vec{R}(s,x) \times \pi(x,t), \forall t \in V. \quad (2)$$

Algorithm 1 described below is a variant of the classic forward local push algorithm [2], the difference being that we added negative residuals. As we will see later on, an edge arrival can result in negative residuals — the edge update affects the amount of residuals a vertex have pushed out. During a forward push iteration for vertex u (Step 6 in Algorithm 1), an α fraction of u's residual will be added to u's estimate. For the rest of $(1-\alpha)$ fraction, each out neighbor of u receives an equal proportion of $(1-\alpha)/d^{\text{OUT}}(u)$. Algorithm 1 repeatedly perform forward push iterations, first for vertices with positive residuals and then for vertices with negative residuals, until for every vertex, its residual divided by the outdegree is within $-\varepsilon$ and ε. It's clear that while we work on vertices with negative residuals, no vertices whose residual is positive and below ε will increase above ε.

2.2.2 Reverse push

Given a target node t, the reverse push algorithm maintains an estimate $\overleftarrow{P}(s,t)$ of $\pi(s,t)$, for every node s in G. It also maitains a residual $\overleftarrow{R}(s,t)$ for s. Let $\overleftarrow{P}(t) = \overleftarrow{P}(\cdot,t)$ denote the vector of estimates and $\overleftarrow{R}(t) = \overleftarrow{R}(\cdot,t)$ for the vector of residuals. Similar to forward push, estimates and residuals satisfy an invariant property [1, 18]:

$$\pi(s,t) = \overleftarrow{P}(s,t) + \sum_{x \in V} \pi(s,x) \times \overleftarrow{R}(x,t), \forall s \in V. \quad (3)$$

Algorithm 1 ForwardLocalPush

```
INPUT: $(s, \vec{P}(s), \vec{R}(s), G, \varepsilon)$
 1: while $\max_u \frac{\vec{R}(s,u)}{d^{\text{OUT}}(u)} > \varepsilon$ do
 2:     FwdPush(u)
 3: while $\min_u \frac{\vec{R}(s,u)}{d^{\text{OUT}}(u)} < -\varepsilon$ do
 4:     FwdPush(u)
 5: return $(\vec{P}(s), \vec{R}(s))$
 6: procedure FwdPush($u$)
 7:     $\vec{P}(s,u)$ += $\alpha \times \vec{R}(s,u)$
 8:     for $v \leftarrow u$ do
 9:         $\vec{R}(s,v)$ += $(1-\alpha) \times \vec{R}(s,u)/d^{\text{OUT}}(u)$
10:     $\vec{R}(s,u) \leftarrow 0$
```

In Algorithm 2 below, we've also added negative residuals. A reverse push iteration on vertex u works backward (step 6): an α fraction of u's residuals goes to u's estimate; to push back the other $1-\alpha$ fraction to an in neighbor v of u, it takes into account the outdegree of v, hence the amount is the residual amount times $(1-\alpha)/d^{\text{OUT}}(v)$. Algorithm 2 keeps pushing residuals back from each vertex, until the residual of every vertex is within $-\varepsilon$ and ε. When this happens, it is guaranteed that $|\pi(s,t) - P(s,t)| \leq \varepsilon$ (cf. Theorem 1 in Anderson et al. [1]).

Algorithm 2 ReverseLocalPush

```
INPUT: $(t, \overleftarrow{P}(t), \overleftarrow{R}(t), G, \varepsilon)$
 1: while $\max_u \overleftarrow{R}(u,t) > \varepsilon$ do
 2:     RevPush(u)
 3: while $\min_u \overleftarrow{R}(u,t) < -\varepsilon$ do
 4:     RevPush(u)
 5: return $(\overleftarrow{P}(t), \overleftarrow{R}(t))$
 6: procedure RevPush($u$)
 7:     $\overleftarrow{P}(u,t)$ += $\alpha \times \overleftarrow{R}(u,t)$
 8:     for $v \rightarrow u$ do
 9:         $\overleftarrow{R}(v,t)$ += $(1-\alpha) \times \overleftarrow{R}(u,t)/d^{\text{OUT}}(v)$
10:     $\overleftarrow{R}(u,t) \leftarrow 0$
```

3. DYNAMIC LOCAL PUSH ALGORITHMS

Given a node, we can run push algorithms to initialize its data structures that include the estimates and residuals. If an edge is inserted/deleted, we can obtain a new pair of estimates and residuals, by adjusting them locally at the updated edge using the invariant equations. Since this adjustment can create residuals that gets above ε or below $-\varepsilon$, we then invoke Algorithm 1 (or Algorithm 2) to push residuals from such nodes. While the idea is simple, the analysis of the number of push iterations is quite intricate. For the rest of this section, we describe an approach to obtain dynamic local push algorithms and show how to analyze their running time. We will introduce a dynamic reverse push algorithm in the first part. Then we will introduce a dynamic forward push algorithm in the second part. Finally we will consider how to extend the algorithms to handle node arrivals and other issues.

3.1 Reverse push

Let t be a (target) vertex of G. Let ε be a parameter between 0 and 1. Our goal is to maintain a pair of estimates and residuals, denoted by $\overleftarrow{P}(t)$ and $\overleftarrow{R}(t)$, such that $\|\overleftarrow{R}(t)\|_\infty \leq \varepsilon$. As mentioned in Section 2.2.2, this will ensure that $|\overleftarrow{P}(s,t) - \pi(s,t)| \leq \varepsilon$, for any node $s \in V$. We start with an equivalent formulation of Equation 3.

Lemma 3. *Equation* (3) *implies*

$$\overleftarrow{P}(s,t)+\alpha\cdot\overleftarrow{R}(s,t) = \sum_{x\in N^{\text{OUT}}(s)}(1-\alpha)\times\frac{\overleftarrow{P}(x,t)}{d^{\text{OUT}}(s)}+\alpha\times\mathbf{1}_{s=t},\ \forall s\in V,$$

and vice versa.

Proof. Let $\vec{\pi}^t = \pi(\cdot,t)$ denote the vector with personalized PageRank from every node of G to t. Let $\Pi = (I-(1-\alpha)D^{-1}A)/\alpha$. It follows from the work of Haveliwala [12] that Π is invertible, and $\vec{\pi}^t = \Pi^{-1}\cdot\vec{e}_t$. This implies that the (s,t)-th entry of Π^{-1} is equal to $\pi(s,t)$, since the s-th entry of $\vec{\pi}^t$ is $\pi(s,t)$, Now, we can write Equation (3) in vector form:

$$\begin{aligned}
&\vec{\pi}^t = \overleftarrow{P}(t) + \Pi^{-1}\cdot\overleftarrow{R}(t)\\
\Leftrightarrow\ &\Pi\cdot\vec{\pi}^t = \Pi\cdot\overleftarrow{P}(t) + \overleftarrow{R}(t)\\
\Leftrightarrow\ &\vec{e}_t = \Pi\cdot\overleftarrow{P}(t) + \overleftarrow{R}(t)\\
\Leftrightarrow\ &\overleftarrow{P}(t) + \alpha\times\overleftarrow{R}(t) = (1-\alpha)D^{-1}A\cdot\overleftarrow{P}(t) + \alpha\times\vec{e}_t
\end{aligned}$$

□

Hence it follows if one can maintain the equivalent invariant in Lemma 3, then one obtains a feasible pair of estimates and residuals for the updated graph. And the question becomes how to maintain the invariant between the estimate and the residual vectors. We will only do it for edge insertions, and it's not hard to work out the details for edge deletions using the same approach. Suppose that an edge $u \to v$ is inserted into G. Then the only vertex that doesn't satisfy the invariant in Lemma 3 is u. In this case, it suffices to update $\overleftarrow{R}(u,t)$ without changing any entries of $\overleftarrow{P}(t)$, because the only variable that has been changed is u's outdegree. The precise formula is obtained by calculating u's correct new residual minus its old residual amount. To simplify the expression below, we take out a common factor of $1/\alpha$, which comes from dividing the α multiplier of $\overleftarrow{R}(u,t)$ as in Lemma 3:

$$\begin{aligned}
&\left(\sum_{x\in N^{\text{OUT}}(u)}\frac{(1-\alpha)\times\overleftarrow{P}(x,t)}{d^{\text{OUT}}(u)+1}\right)+\frac{(1-\alpha)\times\overleftarrow{P}(v,t)}{d^{\text{OUT}}(u)+1}\\
&+\alpha\times\mathbf{1}_{u=t}-\overleftarrow{P}(u,t)\\
&-\left(\sum_{x\in N^{\text{OUT}}(u)}\frac{(1-\alpha)\times\overleftarrow{P}(x,t)}{d^{\text{OUT}}(u)}\right)-\alpha\times\mathbf{1}_{u=t}+\overleftarrow{P}(u,t)\\
&=\frac{(1-\alpha)\times\overleftarrow{P}(v,t)}{d^{\text{OUT}}(u)+1}-\sum_{x\in N^{\text{OUT}}(u)}\frac{(1-\alpha)\times\overleftarrow{P}(x,t)}{d^{\text{OUT}}(u)\times(d^{\text{OUT}}(u)+1)}\\
&=\frac{(1-\alpha)\times\overleftarrow{P}(v,t)}{d^{\text{OUT}}(u)+1}-\frac{\overleftarrow{P}(u,t)+\alpha\times\overleftarrow{R}(u,t)-\alpha\times\mathbf{1}_{u=t}}{d^{\text{OUT}}(u)+1}
\end{aligned}$$

Hence the insertion procedure in Algorithm 3 described below correctly generates a pair of estimates and residuals for the updated graph. Similarly for deletions. It is worth mentioning that we haven't described how to handle nodes whose indegree is zero (also known as dangling nodes): since they do not have in neighbors, their residuals cannot be pushed out. We put off this issue until Section 3.3. Another issue is, if one works with undirected graphs, one needs to apply the insert/delete procedure for both direction of an edge.

```
Algorithm 3 UpdateReversePush
INPUT: $(t, \overleftarrow{P}(t), \overleftarrow{R}(t), u, v, G, \varepsilon)$
Require: $G$ is a directed graph. Let $u \to v$ be the previous edge
    update, with $G$ being the updated graph.
 1: Apply Insert/Delete to $\overleftarrow{P}(t)$ and $\overleftarrow{R}(t)$.
 2: return ReverseLocalPush$(t, \overleftarrow{P}(t), \overleftarrow{R}(t), G, \varepsilon)$.
 3: procedure Insert$(u, v)$
 4:     $\overleftarrow{R}(u,t) \mathrel{+}= \frac{(1-\alpha)\times\overleftarrow{P}(v,t)-\overleftarrow{P}(u,t)-\alpha\times\overleftarrow{R}(u,t)+\alpha\times\mathbf{1}_{u=t}}{d^{\text{OUT}}(u)} \times \frac{1}{\alpha}$
 5: procedure Delete$(u, v)$
 6:     $\overleftarrow{R}(u,t) \mathrel{-}= \frac{(1-\alpha)\times\overleftarrow{P}(v,t)-\overleftarrow{P}(u,t)-\alpha\times\overleftarrow{R}(u,t)+\alpha\times\mathbf{1}_{u=t}}{d^{\text{OUT}}(u)} \times \frac{1}{\alpha}$
```

For the rest of this section, our goal is presenting an analysis of Algorithm 3. Based on previous work of Bahmani et al. [5], it is not difficult to observe that the updated amount of residuals from step 4 and 6) should be small in expectation in a random edge permutation model. However, one can observe that there are already a lot of nonzero residuals in $\overleftarrow{R}(t)$. While these residuals have all been reduced below ε, it gets unclear if they couple with the updated amount of residuals. Our intuition is to solve this problem with amortized analysis.

Theorem 4. *Let* $\langle G_i = (V_i, E_i)\rangle$ *be a sequence of* $k+1$ *graphs such that each graph is obtained from the previous graph with one edge update. Let* $\bar{d}$ *denote the average degree of* G_0. *Let* t *be a random vertex of* G_0. *Then the total running time of maintaining a reverse push solution* $\overleftarrow{P}_i(t)$ *for each graph* G_i *such that* $|\overleftarrow{P}_i(s,t) - \pi_i(s,t)| \leq \varepsilon$, *for any* $s \in V_i$, *using Algorithm 2 and 3, is at most* $O(k/\alpha + k/(n\varepsilon\alpha^2)+\bar{d}/(\alpha\varepsilon))$ *for the following two dynamic graph models:*

- *Arbitrary edge updates of an undirected graph.*
- *Random edge permutation of a directed graph, under the assumption that the indegree of every vertex of* G_0 *is at least one, and no new nodes are added during the edge arrivals.*

Remark: the assumptions under the case of directed graphs is because our proof does not deal with issues of bounding the running time of handling dangling nodes, and this can happen a node of G_0 has no in neighbors, or when a new node arrives, thus creating a node without in neighbors.

We prove this theorem in three steps. First of all, we derive a bound on the running time of Algorithm 2. Secondly, we present a bound on the running time of Algorithm 3 and derive the total running time. Finally, we bound the total running time using properties of the graph model. Due to space limit, we only include the proofs for the case of undirected graphs, and refer the reader to the full version for proofs of the other case.

We start with the running time of Algorithm 2. Lemma 5 below is a Corollary of Theorem 2 in the work of Lofgren et al. [18], the difference being that we subtracted a part that corresponds to the total cost of pushing all the remaining residuals, since these residuals have not yet been pushed out. As we will see later on, this part naturally arises in dynamic settings. Let

$$(\overleftarrow{P}_0(t), \overleftarrow{R}_0(t)) \triangleq \textsc{ReverseLocalPush}(t, \vec{e}_t, \mathbf{0}, G_0, \varepsilon),$$

and

$$\vec{\Phi}_i(x) \triangleq \sum_{s \in V_i} d_i^{\text{IN}}(s) \times \pi_i(s, x), \text{ and} \tag{4}$$

$$\vec{\Phi}_i \triangleq \vec{\Phi}_i(\cdot), \text{ its vector form} \tag{5}$$

Lemma 5. *The running time of Algorithm 2 is at most:*

$$\frac{\vec{\Phi}_0(t) - \|\vec{\Phi}_0 \cdot \overleftarrow{R}_0(t)\|_1}{\alpha\varepsilon}. \tag{6}$$

Proof. To see this, note that every time a node pushes, its estimate increases by $\alpha\varepsilon$, hence the total cost of Algorithm 2 is bounded by:

$$\begin{aligned} &\sum_{s \in V_i} \frac{d_0^{\text{IN}}(s) \times \overleftarrow{P}_0(s,t)}{\alpha\varepsilon} \\ =&\frac{\sum_{s \in V_i} d_0^{\text{IN}}(s) \times (\pi_0(s,t) - \sum_{x \in V_i} \pi_0(s,x) \times \overleftarrow{R}(x,t))}{\alpha\varepsilon} \\ =&\frac{\vec{\Phi}_0(t) - \|\vec{\Phi}_0 \cdot \overleftarrow{R}_0(t)\|_1}{\alpha\varepsilon} \end{aligned}$$

□

Now consider the i-th edge update, for $i = 1, \ldots, k$. That is, we have had the updated graph G_i, after updating G_{i-1} with $e_i = u_i \to v_i$. Then we run Algorithm 3 to update $\overleftarrow{P}_{i-1}(t)$ and $\overleftarrow{R}_{i-1}(t)$. Let

$$\begin{aligned} &(\overleftarrow{P}_i(t), \overleftarrow{R}_i(t)) \triangleq \\ &\quad \textsc{UpdateReversePush}(t, \overleftarrow{P}_{i-1}, \overleftarrow{R}_{i-1}(t), u_i, v_i, G_i, \varepsilon). \end{aligned}$$

For simplicity, let $\Delta_i(t)$ denote the updated amount of residuals for this edge update. That is, if we are inserting e_i, then from step 4 of Algorithm 3, $\Delta_i(u_i, v_i, t)$ is defined to be $1/\alpha$ times:

$$\left| \frac{(1-\alpha) \times \overleftarrow{P}_{i-1}(v_i, t) - \overleftarrow{P}_{i-1}(u_i, t) - \alpha \times \overleftarrow{R}_{i-1}(u_i, t) + \alpha \times \mathbf{1}_{u_i = t}}{d_i^{\text{OUT}}(u_i)} \right|$$

and it could be similarly defined for deletion.

Lemma 6 is the heart of our amortized analysis. The key observation is that one can take care of the cost of an edge update, by comparing the amount of residuals that we pushed out, to the amount of new residuals that get created. Note that the difference between these two masses is precisely the amount of mass that has been received into the estimates. Another useful property of push algorithms is monotonicity: this property has played a crucial role in all the running time analysis of push algorithms. As long as we only push positive residuals or only negative residuals, then the estimates will only change in one way but not the other. This ensures that we could use estimates as a potential function to bound the number of push operations a vertex does.

Lemma 6. *The running time of Algorithm 3 for updating $\overleftarrow{P}_{i-1}(t)$ and $\overleftarrow{R}_{i-1}(t)$ for G_i is at most*

$$\begin{aligned} &\frac{\vec{\Phi}_i(u_i) \times \Delta_i(u_i, v_i, t)}{\alpha\varepsilon} + \frac{\|\vec{\Phi}_i - \vec{\Phi}_{i-1}\|_1}{\alpha} \\ &+ \frac{\|\vec{\Phi}_{i-1} \cdot \overleftarrow{R}_{i-1}(t)\|_1 - \|\vec{\Phi}_i \cdot \overleftarrow{R}_i(t)\|_1}{\alpha\varepsilon} \end{aligned}$$

Proof. Let $\overleftarrow{P}'(t)$ and $\overleftarrow{R}'(t)$ denote the updated values of $\overleftarrow{P}_{i-1}(t)$ and $\overleftarrow{R}_{i-1}(t)$ after step 1 in Algorithm 3. Note that when we invoke Algorithm 2 to reduce the maximum residual of $\overleftarrow{R}'(t)$, we first work on positive residuals and then work on negative residuals. We bound the cost of the two phases separately.

We first bound the cost of pushing out positive residuals (step 1 and 2 in Algorithm 2). Let $\overleftarrow{P}''(t)$ and $\overleftarrow{R}''(t)$ denote the estimate and residual vector after step 2. Since only positive residuals are pushed out, $\overleftarrow{P}''(s,t) \geq \overleftarrow{P}'(s,t)$, for any $s \in V_i$. Hence the cost of this reverse local push is at most:

$$T^+ \triangleq \sum_{s \in V_i} \frac{d_i^{\text{IN}}(s) \times (\overleftarrow{P}''(s,t) - \overleftarrow{P}'(s,t))}{\alpha\varepsilon} \tag{7}$$

By Equation (3),

$$\overleftarrow{P}''(s,t) = \pi_i(s,t) - \sum_{x \in V_i} \pi_i(s,x) \times \overleftarrow{R}''(x,t)$$

Similarly,

$$\overleftarrow{P}'(s,t) = \pi_i(s,t) - \sum_{x \in V_i} \pi_i(s,x) \times \overleftarrow{R}'(x,t)$$

Hence the difference is:

$$\sum_{x \in V_i} \pi_i(s,x) \times (\overleftarrow{R}'(x,t) - \overleftarrow{R}''(x,t))$$

Apply the above expression into Equation (7), we get:

$$\begin{aligned} T^+ &= \sum_{s \in V_i} \sum_{x \in V_i} \frac{d_i^{\text{IN}}(s) \times \pi_i(s,x) \times (\overleftarrow{R}'(x,t) - \overleftarrow{R}''(x,t))}{\alpha\varepsilon} \\ &= \sum_{x \in V_i} \frac{\vec{\Phi}_i(x) \times (\overleftarrow{R}'(x,t) - \overleftarrow{R}''(x,t))}{\alpha\varepsilon} \end{aligned}$$

Here we used $\vec{\Phi}_i(x)$ to simplify the above expression. Now we show that the above expression is at most:

$$T^+ \leq \frac{\|\vec{\Phi}_i \times \overleftarrow{R}'(t)\|_1 - \|\vec{\Phi}_i \cdot \overleftarrow{R}''(t)\|_1}{\alpha\varepsilon}$$

To check this, we compare each vertex $x \in V_i$ and their corresponding entries in the above expression. Since $\vec{\Phi}_i(x)$ is positive for every $x \in V_i$, if $\overleftarrow{R}''(x,t) \geq 0$, then clearly

$$\begin{aligned} &\vec{\Phi}_i(x) \times (\overleftarrow{R}'(x,t) - \overleftarrow{R}''(x,t)) \\ \leq&\vec{\Phi}_i(x) \times (|\overleftarrow{R}'(x,t)| - |\overleftarrow{R}''(x,t)|) \end{aligned}$$

If $\overleftarrow{R}''(x,t) < 0$, then we infer that x has not performed any push operation: otherwise $\overleftarrow{R}''(x,t)$ should be nonnegative. Note that x only received positive residual updates during this phase of forward push. This shows $\overleftarrow{R}'(x,t) \leq \overleftarrow{R}''(x,t) < 0$. Therefore:

$$\begin{aligned} &\vec{\Phi}_i(x) \times (\overleftarrow{R}'(x,t) - \overleftarrow{R}''(x,t)) \\ \leq&0 \leq \vec{\Phi}_i(x) \times (|\overleftarrow{R}'(x,t)| - |\overleftarrow{R}''(x,t)|) \end{aligned}$$

We then bound the cost of pushing out negative residuals (step 3 and 4 in Algorithm 2). Since only negative residuals are pushed out, $\overleftarrow{P}_i(s,t) \leq \overleftarrow{P}''(s,t)$, for any $s \in V_i$. By the same argument we used to derive Equation (7), the total cost of this reverse local push is at most:

$$T^- \triangleq \sum_{x \in V_i} \frac{\vec{\Phi}_i(x) \times (\overleftarrow{R}_i(x,t) - \overleftarrow{R}''(x,t))}{\alpha\varepsilon} \tag{8}$$

We claim that:

$$T^{-} \leq \frac{\|\vec{\Phi}_i \cdot \overleftarrow{R}''(t)\|_1 - \|\vec{\Phi}_i \cdot \overleftarrow{R}_i(t)\|_1}{\alpha\varepsilon} \tag{9}$$

To see this, we compare each vertex $x \in V_i$ and their corresponding entries between Equation (8) and (9). If $\overleftarrow{R}_i(x,t) \leq 0$, then clearly:

$$\begin{aligned}&\vec{\Phi}_i(x) \times (\overleftarrow{R}_i(x,t) - \overleftarrow{R}''(x,t))\\ \leq&\vec{\Phi}_i(x) \times (|\overleftarrow{R}''(x,t)| - |\overleftarrow{R}_i(x,t)|)\end{aligned}$$

If $\overleftarrow{R}_i(x,t) > 0$, then we infer that x does not push out any negative residuals, otherwise $\overleftarrow{R}_i(x,t)$ will be non-positive. This further implies that $\overleftarrow{R}''(x,t) \geq \overleftarrow{R}_i(x,t) > 0$, since x may only receive negative residual updates during this phase. Therefore,

$$\begin{aligned}&\vec{\Phi}_i(x) \times (\overleftarrow{R}_i(x,t) - \overleftarrow{R}''(x,t))\\ &\leq 0 \leq \vec{\Phi}_i(x) \times (|\overleftarrow{R}''(x,t)| - |\overleftarrow{R}_i(x,t)|))\end{aligned}$$

Finally, we obtain a bound on the total running time of invoking Algorithm 2 to reduce the maximum residual:

$$T \triangleq T^{+} + T^{-} \leq \|\vec{\Phi}_i \cdot \overleftarrow{R}'(t)\|_1 - \|\vec{\Phi}_i \cdot \overleftarrow{R}_i(t)\|_1 \tag{10}$$

We work on the above equation to finish the proof. First, since $\overleftarrow{R}'(t)$ is $\overleftarrow{R}_{i-1}(t)$ with an update of $\Delta_i(u_i, v_i, t)$ on vertex u_i, we can separate out $\Delta_i(u_i, v_i, t)$ from $\overleftarrow{R}'(t)$:

$$\|\vec{\Phi}_i \cdot \overleftarrow{R}'(t)\|_1 \leq \vec{\Phi}_i(u_i) \times \Delta_i(u_i, v_i, t) + \|\vec{\Phi}_i \cdot \overleftarrow{R}_{i-1}(t)\|_1$$

Applying the above equation into Equation (10), we found that

$$T \leq \frac{\|\vec{\Phi}_i \cdot \overleftarrow{R}_{i-1}(t)\|_1 + \vec{\Phi}_i(u_i) \times \Delta_i(u_i, v_i, t) - \|\vec{\Phi}_i \cdot \overleftarrow{R}_i(t)\|_1}{\alpha\varepsilon} \tag{11}$$

Then, since $\|\overleftarrow{R}_{i-1}(t)\|_\infty$ is at most ε,

$$\begin{aligned}\|\vec{\Phi}_i \cdot \overleftarrow{R}_{i-1}(t)\|_1 \leq&\|(\vec{\Phi}_i - \vec{\Phi}_{i-1}) \cdot \overleftarrow{R}_{i-1}(t)\|_1 + \|\vec{\Phi}_{i-1} \cdot \overleftarrow{R}_{i-1}(t)\|_1\\ \leq&\varepsilon \times \|\vec{\Phi}_i - \vec{\Phi}_{i-1}\|_1 + \|\vec{\Phi}_{i-1} \cdot \overleftarrow{R}_{i-1}(t)\|_1\end{aligned}$$

Applying the above equation into equation (11) gives us the desired conclusion. □

Lemma 6 naturally suggests that there is a way to amortize per edge update costs, by cancelling out the weighted residual terms. Hence, by summing up the initialization cost in Lemma 5, and the edge update cost in Lemma 6, for $i = 1, \ldots, k$, we obtained the total cost of maintaining the estimates and residuals for every graph G_i, from $i = 0, \ldots, k$:

$$\psi(t) \triangleq \frac{\vec{\Phi}_0(t)}{\alpha\varepsilon} + \sum_{i=1}^{k} \frac{\vec{\Phi}_i(u_i) \times \Delta_i(u_i, v_i, t)}{\alpha\varepsilon} + \sum_{i=1}^{k} \frac{\|\vec{\Phi}_i - \vec{\Phi}_{i-1}\|_1}{\alpha}$$

Now we are ready to present an average case analysis, by summing up $\psi(t)$ over all the target vertices. First, we know from the work of Lofgren et al. (Theorem 1, [18]) that:

$$\sum_{t \in V_0} \vec{\Phi}_0(t) = m.$$

Therefore, the total running time of maintaining a pair of estimates and residuals with threshold ε for every node $t \in V_0$, starting from G_0 with m edges, during k edge updates, is at most:

$$\begin{aligned}\Psi \triangleq \sum_{t \in V_0} \psi(t) = \frac{m}{\alpha\varepsilon} + \sum_{i=1}^{k} \sum_{t \in V_0} \frac{\vec{\Phi}_i(u_i) \times \Delta_i(u_i, v_i, t)}{\alpha\varepsilon} \quad (12)\\ + \sum_{i=1}^{k} \left(\sum_{t \in V_0} \frac{\|\vec{\Phi}_i - \vec{\Phi}_{i-1}\|_1}{\alpha} \right) \quad (13)\end{aligned}$$

where we have changed the order of summation for the second and third term.

Lemma 7. *Let t be any vertex of V_i. Then*

$$\sum_{t \in V_i} \Delta_i(u_i, v_i, t) \leq (2n\varepsilon + 2)/(\alpha \times d_i^{\text{OUT}}(u_i)),$$

for any $i = 1, \ldots, k$.

Proof. We deal with the sum of numerators of $\Delta_i(u_i, v_i, t)$ first, since the denominator of $\Delta_i(u_i, v_i, t)$ does not depend on t. We first take out each term from the absolute value to obtain an upper bound of $\Delta_i(u_i, v_i, t)$:

$$\begin{aligned}&(1-\alpha) \times \overleftarrow{P}_{i-1}(v_i, t) + \overleftarrow{P}_{i-1}(u_i, t)\\ &+ \alpha \times \overleftarrow{R}_{i-1}(u_i, t) + \alpha \times \mathbf{1}_{u_i = t}\\ \leq&\ (1-\alpha) \times (\pi_{i-1}(v_i, t) + \varepsilon) + (\pi_{i-1}(u_i, t) + \varepsilon)\\ &+ \alpha \times \varepsilon + \alpha \times \mathbf{1}_{u_i = t}\end{aligned}$$

We used the fact that $\overleftarrow{P}_{i-1}(u_i, t) \leq \pi_{i-1}(u_i, t) + \varepsilon$ (similarly for v_i) and $\overleftarrow{R}_{i-1}(u_i, t) \leq \varepsilon$. If we sum up the above expression over $t \in V_i$, and take the denominator back to the expression, it leads to the following much simplified conclusion:

$$\sum_{t \in V_i} \Delta_i(u_i, v_i, t) \leq \frac{2n\varepsilon + 2}{\alpha \times d_i^{\text{OUT}}(u_i)}$$

□

Arbitrary edge update on an undirected graph.

Proof of Theorem 4, Part 1: Note that Lemma 5 (initialization cost) holds for undirected graphs as well. When we update an edge, We need to take into account that we applied insertion/deletion twice (step 4 and 6 in Algorithm 3), for each direction of the edge. This makes a difference when we separate out the difference between $\overleftarrow{R}'(t)$ and $\overleftarrow{R}_{i-1}(t)$. Hence, it suffices to add

$$\frac{\vec{\Phi}_i(v_i) \times \Delta_i(v_i, u_i, t)}{\alpha\varepsilon}$$

into the bound of Lemma 6: the rest of the proof holds for undirected graphs. In conclusion, we found that the total running time of maintaining $\overleftarrow{P}_i(t)$ and $\overleftarrow{R}_i(t)$ is at most:

$$\begin{aligned}\Psi \triangleq \frac{m}{\alpha\varepsilon} + \sum_{i=1}^{k} \sum_{t \in V_0} \frac{\vec{\Phi}_i(u_i) \times \Delta_i(u_i, v_i, t) + \vec{\Phi}_i(v_i) \times \Delta_i(v_i, u_i, t)}{\alpha\varepsilon}\\ + \sum_{i=1}^{k} \sum_{t \in V_0} \frac{\|\vec{\Phi}_i - \vec{\Phi}_{i-1}\|_1}{\alpha}\end{aligned}$$

Now we claim that with $\vec{\Phi}_i(x) = d_i(x)$, for any $x \in V_i$ and $i = 0, \ldots, k$. By Proposition 1,

$$\vec{\Phi}_i(x) = \sum_{s \in V_i} d_i(s) \times \pi_i(s, x) = \sum_{s \in V_i} d_i(x) \times \pi_i(x, s). = d_i(x).$$

Combined with Lemma 7, the second term of Φ is at most $k \times (\frac{4n}{\alpha^2} + \frac{4}{\alpha^2 \varepsilon})$.

It also follows that $\vec{\Phi}_i$ is equal to the degree vector of G_i, and the l_1 difference between $\vec{\Phi}_i$ and $\vec{\Phi}_{i-1}$ is equal to 2, since only the degrees of u_i and v_i changed by one.

To sum up,

$$\Psi \leq \frac{m}{\alpha\varepsilon} + k \times (\frac{4n}{\alpha^2} + \frac{4}{\alpha^2\varepsilon}) + \frac{2nk}{\alpha}.$$

3.2 Forward push

Now we apply a similar approach to derive a dynamic forward push algorithm. Let s be a source vertex of G. Let ε be a threshold parameter between 0 and 1. Our goal is to maintain a pair of estimates $\vec{P}(s)$ and residuals $\vec{R}(s)$ such that $\vec{R}(s,t)/d^{\text{OUT}}(t) \leq \varepsilon$, for any $t \in V$. We begin by describing an equivalent invariant property to Equation 2.

Lemma 8. *Equation 2 implies*

$$\vec{P}(s,t) + \alpha \times \vec{R}(s,t) = \sum_{x \in N^{\text{IN}}(t)} \frac{\vec{P}(s,x)}{d^{\text{OUT}}(x)} + \alpha \times \mathbf{1}_{t=s}, \forall t \in V,$$

and vice versa.

Proof. Let $\vec{\pi}_s = \pi(s,\cdot)$ denote a vector with personalized PageRank from s to every node of G. Let $\Pi = \alpha \times (I - (1-\alpha)A^{\intercal}D^{-1})^{-1}$: the fact that Π exists follows from Definition 1, and $\vec{\pi}_s$ is equal to $\Pi \cdot \vec{e}_s$. Therefore equation 2 in vector form is equivalent to:

$$\begin{aligned}
&\vec{\pi}_s = \vec{P}(s) + \Pi \cdot \vec{R}(s) \\
\Leftrightarrow\ &\Pi^{-1} \cdot \vec{\pi}_s = \Pi^{-1}\vec{P}(s) + \vec{R}(s) \\
\Leftrightarrow\ &\vec{e}_s = \frac{I - (1-\alpha)A^{\intercal}D^{-1}}{\alpha} \cdot \vec{P}(s) + \vec{R}(s) \\
\Leftrightarrow\ &\vec{P}(s) + \alpha \times \vec{R}(s) = (1-\alpha)A^{\intercal}D^{-1}\vec{P}(s) + \alpha \times \vec{e}_s
\end{aligned}$$

□

Now we use Lemma 8 to derive the update procedure. Consider when an edge $u \to v$ is inserted to G. Since the outdegree of u increases by 1, the invariant does not hold for the out neighbors of u any more: the reason being that every out neighbor receives an equal proportion of mass from u which is $(1-\alpha)$ divided by the outdegree of u times the amount of mass that u pushed. To solve this imbalance, clearly a simple solution is to scale $\vec{P}(s,u)$ by a factor of $(d^{\text{OUT}}(u)+1)/d^{\text{OUT}}(u)$. This ensures the invariant for every out neighbor of u, except v. This is because we need to take into account the amount of residuals v should have received previously, had the edge $u \to v$ existed. Finally, since we have increased $\vec{P}(s,u)$, this will break the invariant for u. Hence we will reduce $\vec{R}(s,u)$ by the increased amount on $\vec{P}(s,u)$, divided by α. See Algorithm 4 below for details. To work with an undirected graph, we will apply insert/delete twice, for both direction between u and v. It's not hard to see this, following our discussion above.

Next we prove a theorem on the update cost of Algorithm 4. We will prove it on undirected graphs. It's not clear to us how to analyze directed graphs with forward push: the difficulty being that there is no clean bound on the error of Algorithm 1, between the estimates and the true personalized PageRank value.

Algorithm 4 UpdateForwardPush

```
INPUT: $(s, \vec{P}(s), \vec{R}(s), u, v, G, \varepsilon)$
Require: $G$ is a directed graph. Let $u \to v$ be the previous edge
    update, with $G$ being the updated graph.
 1: Apply Insert/Delete to $\vec{P}(s)$ and $\vec{R}(s)$.
 2: return ForwardLocalPush$(s, \vec{P}(s), \vec{R}(s), G, \varepsilon)$
 3: procedure Insert$(u, v)$
 4:     $\vec{P}(s,u) \mathrel{*}= \frac{d^{\text{OUT}}(u)}{d^{\text{OUT}}(u)-1}$
 5:     $\vec{R}(s,u) \mathrel{-}= \frac{\vec{P}(s,u)}{d^{\text{OUT}}(u)} \cdot \frac{1}{\alpha}$
 6:     $\vec{R}(s,v) \mathrel{+}= \frac{(1-\alpha)\times\vec{P}(s,u)}{d^{\text{OUT}}(u)} \cdot \frac{1}{\alpha}$
 7: procedure Delete$(u, v)$
 8:     $\vec{P}(s,u) \mathrel{*}= \frac{d^{\text{OUT}}(u)}{d^{\text{OUT}}(u)+1}$
 9:     $\vec{R}(s,u) \mathrel{+}= \frac{\vec{P}(s,u)}{d^{\text{OUT}}(u)} \cdot \frac{1}{\alpha}$
10:     $\vec{R}(s,v) \mathrel{-}= \frac{(1-\alpha)\times\vec{P}(s,u)}{d^{\text{OUT}}(u)} \cdot \frac{1}{\alpha}$
```

Theorem 9. *Let $\langle G_i = (V_i, E_i)\rangle$ be a sequence of $k+1$ undirected graphs, such that each graph is obtained from the previous graph by one edge update. Let s be a random vertex of V_0. And let ε be a parameter between 0 and 1. Then the total running time of maintaining a forward local push solution $\vec{P}_i(s)$ for each graph G_i such that $|\vec{P}_i(s,t) - \pi_i(s,t)|/d_i(t) \leq \varepsilon$, for any $t \in V_i$, using Algorithms 4 is at most $O(k/(\alpha^2) + k/(n\alpha^2\varepsilon) + 1/(\alpha\varepsilon))$.*

As long as $|\vec{R}_i(s,t)|/d_i(t) \leq \varepsilon$, for any $t \in V_i$ and $i = 0, \ldots, k$, then $|\vec{P}_i(s,t) - \pi_i(s,t)|/d_i(t) \leq \varepsilon$. This accuracy guarantee follows from the work of Anderson et al. and Lofgren et al. [2, 15]. Hence, we will focus on analyzing running time. To derive this theorem, we divide the arguments into three parts: first, we present a bound on the initiation cost as well as the update cost per edge; secondly, we amortize the costs together, cancelling out the residual terms; finally, we use properties from undirected graphs to bound the total cost. The proof can be found in the full version of this paper.

3.3 Discussions

The algorithms presented in the previous sections do not handle dangling nodes. We consider how to take care of this situation. We also extend the algorithms to handle newly arrived nodes.

Dangling nodes.

There are two possible ways to handle dangling nodes. The first way is to create a separate sink node. Consider the case of doing reverse push (similarly for forward push). If a node u does not have any in neighbors and need to perform a push operation, then the amount of mass will be pushed to the sink node. These mass will stay at the sink node. Another way is to insert an edge from t to u, if t is the target node for maintaining reverse push solutions. Later on if an edge is added from v to u, so that u is not a dangling node anymore, then we first delete the edge from t to u and then insert an edge from v to u.

Node arrivals.

When a new node arrives, one can insert the edges one by one using the procedures in Algorithm 3 (line 3) and Algorithm 4 (line 3), and then invoke the local push algorithms to reduce the maximum absolute values of residuals. Similarly for node deletions.

4. NUMERICAL RESULTS

In this section, we evaluate our approach in experiments. We first compare our dynamic forward local push algorithm (Algorithm 4) to the previous work of Ohsaka et al. [20]. Our dynamic Forward Push algorithm differs from Ohsaka et. al.'s in an important way. When an edge (u, v) arrives, they propose to push the change in residual immediately to all of u's neighbors, requiring $\Omega(d(u))$ time, in addition to any pushes needed to restore the invariant. In contrast, we modify the estimate and residual values only at u and v, taking only $O(1)$ time, before performing any pushes needed to restore the invariant. We found that this simple optimization decreased the number of residual values updated and led to a 1.5 - 3.5 times improvement in running time, without sacrificing accuracy.

We then make a comparison to the random walk approach of Bahmani et al. [5]. We consider the problem of identifying the top-K vertices that have the highest personalized PageRank from a source node, with random walks being commonly used to solve it on social networks [11, 19]. We found that the random walk algorithm uses 4.5 to 12 times as much storage compared to forward local push algorithm, and requires 1.5 to 2.5 times as much time to update as well, to achieve the same level of precision.

Finally, we evaluate our dynamic reverse push algorithm. We compare against the baseline where we recompute the reverse push from scratch after every edge update. We found that our approach is 100x faster than this baseline.

4.1 Experimental setup

We implement the experiments in Scala. We use an Amazon EC2 Ubuntu machine with 64GB of RAM and 16 processors of Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz. We tested with three undirected social networks graphs, all downloaded from https://snap.stanford.edu/data/. The statistics are listed in Table 1 below. For each social network, we generate a uniformly random edge stream of the graph. Given a test vertex, we initialize its data structures using the subgraph that consists of the first half of the edges in the random edge stream. After that, for each new edge arrival, we update its data structure depending the approach we are using. The teleport probability α is set to 0.2 in all the experiments. For ease of comparison, we use LazyFwdUpdate to refer to Algorithm 4 (because it "lazily" avoids pushing from vertices that receive new edges), Ohsaka et al.'s Algorithm TrackingPPR and the random walk update Algorithm by RandomWalk.

graph	# nodes	# edges
DBLP	317,080	1,049,866
LiveJournal	3,997,962	34,681,189
Orkut	3,072,441	117,185,083

Table 1: Basic statistics of graphs that we experimented with.

4.2 Residual updates and running time

We compare LazyFwdUpdate and TrackingPPR over 100 uniformly sampled source vertices from each graph. In our implementation, we first update the previous estimate vectors and residual vectors based on LazyFwdUpdate and TrackingPPR, respectively. We then invoke the same forward local push routine (Algorithm 1) to reduce any residual that is outside the desired threshold. At the end of the edge stream, we compare the performance of each algorithm, using measurements described below:

- Residual update: the number of times that a vertex's residual is updated. Each residual update corresponds to an update in the priority queue of residuals.
- Push iterations: the number of times that Algorithm 1 performs a push operation in step 6. The cost of a push iteration is the degree of the pushed vertex times the cost of a residual update.
- Running time: the amount of update time it takes to maintain estimates and residuals. We exclude the amount of time it takes to initialize the data structure. Note that this cost is negligible compared to the amount of update time in our experiment.
- l_1 error: the l_1 distance between the computed estimates and the benchmark values. To get the benchmark, we ran the local push algorithm with threshold $0.02/n$, where n is the number of vertices of the graph.
- Storage: the number of nonzero estimates and residuals maintained.

For residual update, push iteraions and running time, we computed the average value over the 100 samples. For l_1 error, we compute the median error over the 100 samples. The test results is shown in Table 2. We found that TrackingPPR does more residual updates than LazyFwdUpdate, for all three graphs, and the improvement is more significant for denser graphs. While Algorithm 4 does more push operations than TrackingPPR, since the latter already does a push operation before invoking Algorithm 1, the compound effect is that we achieved 1.5 to 5 times speed up, without sacrificing accuracy.

4.3 Comparison to random walks

We compare LazyFwdUpdate and RandomWalk for the problem of finding top-K vertices ranked by their personalized Pagerank from a source vertex. We sample 100 test vertices uniformly at random. For each test vertex, the correct top-K solution is the set of K vertices with highest personalized Pagerank from that vertex. The performance of each algorithm is measured by:

- Accuracy: the number of correctly identified vertices (among its computed K vertices with highest personalized Pagerank) divided by K.
- Running time: the amount of time to maintain/update the data structures.
- Storage: to compare storage used by RandomWalk, we multiply the number of random walks by the expected length of a walk (5 in our experiment) times 4 (number of bytes to store an integer vertex ID). [1] For LazyFwdUpdate, we multiply the number of nonzero

[1] To allow for efficient update, it's necessary to build an inverted index for each vertex that quickly finds to the set of random walks crossing it. In our implementation we built a hashmap for this purpose, resulting in an additional fac-

	DBLP		LiveJournal		Orkut	
	LazyFwd Update	Tracking PPR	LazyFwd Update	Tracking PPR	LazyFwd Update	Tracking PPR
Residual updates	1.5×10^6	2.4×10^6	2.5×10^6	1.8×10^7	2.9×10^6	6.2×10^7
Push iterations	2.3×10^5	1.8×10^5	1.5×10^5	1.0×10^5	5.3×10^4	2.6×10^4
Running time	$2.9s$	$4.7s$	17.1	51.4	49.8	176.4
l_1 error	0.031	0.031	0.145	0.150	0.238	0.244

Table 2: Test results between LazyFwdUpdate with $\varepsilon = 7 \times 10^{-7}$ and TrackingPPR with $\varepsilon = 10^{-6}$. For each graph, we sampled 100 vertices uniformly at random.

	DBLP		LiveJournal		Orkut	
	LazyFwd Update	Random Walk	LazyFwd Update	Random Walk	LazyFwd Update	Random Walk
Storage	2.6×10^4	3.2×10^5	5.3×10^4	3.2×10^5	6.3×10^4	3.2×10^5
Accuracy	0.96	0.92	0.88	0.88	0.74	0.78
Running time	0.32s	0.88s	8.8s	15.3s	29.7s	53.0s

Table 3: Test results for top-50 between LazyFwdUpdate with $\varepsilon = 5 \times 10^{-5}$ and RandomWalk with 16000 walks. For each graph, we sampled 100 vertices uniformly at random. The parameters are chosen so that the median accuracy of both algorithms are around 0.9 on LiveJournal.

	DBLP		LiveJournal		Orkut	
	LazyFwd Update	Random Walk	LazyFwd Update	Random Walk	LazyFwd Update	Random Walk
Storage	6.0×10^4	5.0×10^5	1.2×10^5	5.0×10^5	1.5×10^5	5.0×10^5
Accuracy	0.96	0.91	0.91	0.88	0.75	0.80
Running time	0.41s	1.24s	8.6s	16.0s	30.1s	59.7s

Table 4: Test results for top-100 between LazyFwdUpdate with $\varepsilon = 2 \times 10^{-5}$ and RandomWalk with 25000 walks. For each graph, we sampled 100 vertices uniformly at random. The parameters are chosen so that the median accuracy of both algorithms are around 0.9 on LiveJournal.

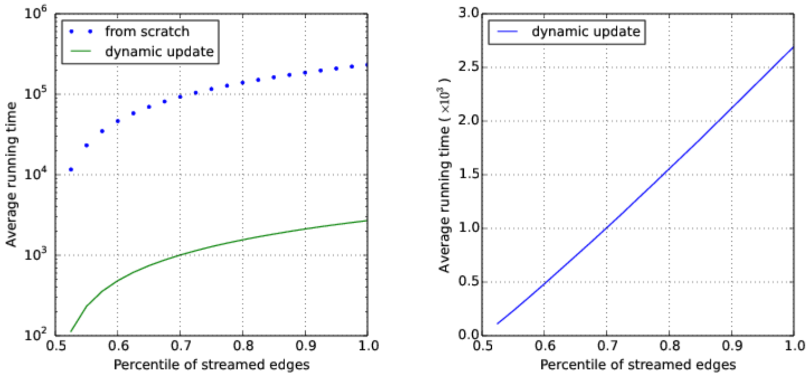

Figure 1: Running time comparisons on LiveJournal, between maintaining reverse push using Algorithm 3 and computing reverse push from scratch for every edge insertion with $\varepsilon = 10^{-4}$. We take 100 random test nodes and compute the average running time for each algorithm.

estimates and residuals by 8 (number of bytes to store a pair of integer and floating point number).

Table 3 and 4 below describe the results with K being 50 and 100, respectively. We take the average of running time and storage, and median of accuracy over the 100 sampled vertices. For the problem of identifying top-50 nodes, we found that RandomWalk uses 4.5 to 12 times as much storage compared to LazyFwdUpdate, and uses 1.6 to 2.7 as much running time. On Orkut graph, when accuracy is controlled to be at the same level as RandomWalk for LazyFwdUpdate (with $\varepsilon = 4 \times 10^{-5}$), the average amount of storage becomes 7.0×10^4 and the average running time becomes $33.3s$. When K is 100, the test results are qualitatively consistent with the above conclusion.

tor of 2 in the amount of storage used. Since there might be more efficient implementation, we do not take this additional factor into account in our comparison.

4.4 Evaluation of dynamic reverse push algorithm

In this section we evaluate the performance of our dynamic reverse push algorithm. We compare our proposed solution (Algorithm 3) to the baseline, where we recompute reverse push from scratch after every edge insertion using Algorithm 2. We sample 100 target vertices uniformly at random and take the average of the running time over these 100 samples. We have chosen the maximum residual parameter $\varepsilon = 10^{-4}$ so that at the end of the edge stream, the median l_1 error over the 100 samples between the maintained reverse push estimates and the true values is less than 0.1. Because it takes too long to compute reverse push from scratch after every edge insertion for 100 sampled vertices, we only computed reverse push from scratch for the first 1000 edge insertions. We take the average running time for performing reverse push during 1000 edge updates, and use this average value times the number of inserted edges as an approximation to the true running time, if we have computed reverse push from scratch for every edge insertion. Figure 1 shows the experiment result on the LiveJournal graph. We found that our approach is 100x faster than the baseline. The aggregate running time for updating reverse push has a linear correlation with the number of edges added. This is consistent with the theoretical bound from Theorem 4.

Acknowledgement.

Research supported by the DARPA GRAPHS program via grant FA9550-12-1-0411, and by NSF grant 1447697.

5. REFERENCES

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

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

[3] Konstantin Avrachenkov, Nelly Litvak, Danil Nemirovsky, and Natalia Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM Journal on Numerical Analysis*, 2007.

[4] Lars Backstrom and Jure Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*. ACM, 2011.

[5] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized pagerank. *Proceedings of the VLDB Endowment*, 4(3):173–184, 2010.

[6] Shumeet Baluja, Rohan Seth, D Sivakumar, Yushi Jing, Jay Yagnik, Shankar Kumar, Deepak Ravichandran, and Mohamed Aly. Video suggestion and discovery for youtube: taking random walks through the view graph. In *Proceedings of the 17th international conference on World Wide Web*, pages 895–904. ACM, 2008.

[7] Pavel Berkhin. Bookmark-coloring algorithm for personalized pagerank computing. *Internet Mathematics*, 3(1):41–62, 2006.

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

[9] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2005.

[10] David F Gleich. Pagerank beyond the web. *SIAM Review*, 57(3):321–363, 2015.

[11] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2013.

[12] Taher H Haveliwala. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search. *Knowledge and Data Engineering, IEEE Transactions on*, 15(4):784–796, 2003.

[13] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *Proceedings of the 12th international conference on World Wide Web*. ACM, 2003.

[14] Peter Lofgren. On the complexity of the monte carlo method for incremental pagerank. *Information Processing Letters*, 114(3):104–106, 2014.

[15] Peter Lofgren. Efficient algorithms for personalized pagerank. *arXiv preprint arXiv:1512.04633*, 2015.

[16] Peter Lofgren, Siddhartha Banerjee, and Ashish Goel. Bidirectional pagerank estimation: From average-case to worst-case. In *Algorithms and Models for the Web Graph*, pages 164–176. Springer, 2015.

[17] Peter Lofgren, Siddhartha Banerjee, and Ashish Goel. Personalized pagerank estimation and search: A bidirectional approach. In *WSDM*, 2016.

[18] 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. ACM, 2014.

[19] Ioannis Mitliagkas, Michael Borokhovich, Alexandros G Dimakis, and Constantine Caramanis. Frogwild!: fast pagerank approximations on graph engines. *Proceedings of the VLDB Endowment*, 8(8):874–885, 2015.

[20] Naoto Ohsaka, Takanori Maehara, and Ken-ichi Kawarabayashi. Efficient pagerank tracking in evolving networks. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 875–884. ACM, 2015.

[21] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[22] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, page 3. ACM, 2012.