# Fast Incremental PageRank on Dynamic Networks

Zexing Zhan[1,2], Ruimin Hu[1,2(✉)], Xiyue Gao[1,2], and Nian Huai[1,2]

[1] National Engineering Research Center for Multimedia Software, School of Computer Science, Wuhan University, Wuhan 430072, China
hrm@whu.edu.com

[2] Hubei Key Laboratory of Multimedia and Network Communication Engineering, Wuhan University, Wuhan 430072, China

**Abstract.** Real-world networks are very large and are constantly changing. Computing PageRank values for such dynamic networks is an important challenge in network science. In this paper, we propose an efficient Monte Carlo based algorithm for PageRank tracking on dynamic networks. A revisit probability model is also presented to provide theoretical support for our algorithm. For a graph with $n$ nodes, the proposed algorithm maintains only $nR$ random walk segments ($R$ random walks starting from each node) in memory. The time cost to update PageRank scores for each graph modification is proportional to $n/|\boldsymbol{E}|$ ($\boldsymbol{E}$ is the edge set). Experiments on 5 real-world networks indicate that our algorithm is 1.3–30 times faster than state-of-the-art algorithms and does not accumulate any errors.

**Keywords:** PageRank tracking · Monte Carlo · Random walk · Incremental computing · Dynamic networks

## 1 Introduction

PageRank [6] was first used by Google in 1998 to provide better results in their search engine. It measures the popularity of a web page from the topological structure of the Web, independent of the page content. Over the last decades, PageRank has emerged as a very effective measure of reputation for both web graphs and online social networks, which was historically known as eigenvector centrality [5] or Katz centrality [11]. However, real-world networks are very large and are evolving rapidly. For example, there are 60T web pages in the Web, and it grows with more than 600K new pages every second. In this case, PageRank algorithms that work only for static networks are insufficient, especially when it is desirable to track PageRank values in real-time rather than to wait for a batched computation.

This work was supported by the National Natural Science Foundation of China [grant No. U1736206] and the National Key R&D Program of China [grant No. 2017YFC0803700].


M. Bakaev et al. (Eds.): ICWE 2019, LNCS 11496, pp. 154–168, 2019.
https://doi.org/10.1007/978-3-030-19274-7_12

A dynamic network is a sequence of graphs $\{G(t) : t = 0, 1, 2, ...\}$, so that $G(t+1)$ is obtained by inserting an edge to or removing an edge from $G(t)$. To efficiently track PageRank scores on dynamic networks, incremental algorithms are necessary. Currently, there are two main categories of incremental PageRank algorithms: aggregation algorithms and Monte Carlo based algorithms. A detailed description of these algorithms is provided in Sects. 2 and 3. In this paper, we focus on Monte Carlo based algorithms. Existing Monte Carlo based algorithms have two main drawbacks. Firstly, they accumulate errors over lengthy evolution. Both [3] and [14] simply assume that a random walk would never revisit a node. This assumption however is not the truth, which makes the approximation error accumulate quickly on real-world networks. Also, since [14] does not save any random walk segments, its way to adjust previous random walk segments also brings error. Secondly, they are inefficient. For each graph modification, existing algorithms simulate too many unnecessary random walks to update PageRank values. Besides, [3] keeps a duplicate of a random walk segment for every node it is passing through, which makes it really a disaster to maintain them.

To address those two limitations mentioned above, we propose a novel Monte Carlo based algorithm for PageRank tracking on dynamic networks. Our method supposes each edge has its own revisit probability as is often the case in real-world networks. A revisit probability model is presented to provide theoretical support for our algorithm. Besides, our method saves only $nR$ random walk segments ($R$ random walks starting from each node) in memory. An efficient and well structure method is implemented to maintains these random walk segments. Experiments on several real-world networks show that our method is 1.3–30 times faster than state-of-the-art algorithms and does not accumulate any errors in long-term evolution. Our main contributions are as follows:

- Theory: we propose the revisit probability model for analyzing Monte Carlo based PageRank tracking problem.
- Algorithm: we propose an efficient algorithm that also improves the accuracy.
- Experiments: we report experiments on 5 real-world networks, and compare our algorithm to state-of-the-art methods.

The rest of the paper is structured as follows. Section 2 describes some preliminaries for Monte Carlo based algorithms; Sect. 3 surveys related works; Sect. 4 presents the revisit probability model and the proposed algorithm; Sect. 5 reports experimental results; Finally, Sect. 6 concludes this work.

## 2 Preliminaries

Before describing related work, we briefly describe some preliminaries for Monte Carlo based PageRank tracking algorithms.

### 2.1 PageRank

The basic idea of PageRank is that more important pages are likely to receive more links from other pages. That means, the importance of a page depends on the number and quality of links to the page. Hence, the PageRank value $\pi_v$ of page $v$ is computed by taking into account the set of pages $\boldsymbol{in}_v$ pointing to $v$. According to Brin and Page [6]:

$$\pi_v = \alpha \sum_{u \in \boldsymbol{in}_v} \frac{\pi_u}{outdeg_u} + (1 - \alpha). \tag{1}$$

Here $\alpha \in (0, 1)$ is the teleport probability and $outdeg_u$ is the outdegree of page $u$, that is the number of hyperlinks coming out from $u$. When stacking all $\pi_v$ into a vector $\boldsymbol{\pi}$, we get:

$$\boldsymbol{\pi} = \alpha \boldsymbol{P} \boldsymbol{\pi} + (1 - \alpha)\mathbf{1}. \tag{2}$$

where $\mathbf{1} = [1, ..., 1]$ and $\boldsymbol{P} = \{p_{v,u}\}$ is the transition matrix, such that $p_{v,u} = 1/outdeg_u$ if there is a hyperlink from $u$ to $v$ and $p_{v,u} = 0$, otherwise.

### 2.2 Approximating PageRank

The Monte Carlo method [2] approximates PageRank by simulating exactly $R$ random walks starting from each node in a graph. Each of these random walks can be terminated at each step either with probability $\epsilon$ (here we call $\epsilon$ as the reset probability and $\epsilon = 1 - \alpha$), or when it reaches a dangling node. A dangling node is a node that does not contain any out edge. Assume for each node $u$, $V_u$ is the total number of times that all simulated random walks visit $v$. Then, we approximate the PageRank of $v$, denoted by $\pi_u$, with:

$$\widetilde{\pi}_u = \frac{V_u}{nR/\epsilon} \tag{3}$$

where $n$ is the number of nodes in a graph.

### 2.3 Updating PageRank

By storing all simulated random walk segments, the original Monte Carlo method allows to perform continuous update of the PageRank as the structure of the graph changes. However, when an edge $e(u, w)$ is modified at time $t + 1$, it has to adjust all those random walk segments passing through node $u$ at time $t$. Fortunately, [3] proved that a random walk segment needs to be adjusted only if it visits the node $u$ and picks $w$ as the next node. In expectation, the number of times a walk segment visits $u$ is $\frac{\pi_u}{\epsilon}$. For each such visit, the probability for the walk to need a reroute is $\frac{1}{outdeg_u(t+1)}$. And there are a total of $nR$ random walk segments. Define $M_{t+1}$ to be the number of random walk segments that need to be adjusted at time $t + 1$. By union bound, we have:

$$E[M_{t+1}] \leq \frac{nR}{\epsilon} E[\frac{\widetilde{\pi}_u(t)}{outdeg_{u(t+1)}}]. \tag{4}$$

## 3 Related Work

In this section, we review some works for tracking PageRank on evolving networks. A simple way to keep the PageRank scores updated is to recompute the values using the simple power iteration method [6] for each change in the network. But, this can be very costly. For a network that has $n$ nodes, with a reset probability of $\epsilon$, it takes $\Omega(\frac{kn^2}{1/(1-\epsilon)})$ total time to recompute the PageRank values for $k$ edge modifications. Similarly, the Monte Carlo method [2], working by simulating exactly $R$ random walk starting from each node, results in a total $\Omega(\frac{knR}{\epsilon})$ work, which is also inefficient. Therefore, lots of incremental methods were proposed for updating the approximation of PageRank. Based on their core techniques, these methods can be categorized into two general categories:

- Aggregation Algorithms

The basic observation of methods in this category is that evolution of the graph is slow, with large parts of it remaining unchanged. Based on that, several aggregation algorithms were proposed in [7,8,10,19]. When an edge $e(u, w)$ is inserted to or removed from a network at time $t + 1$, these algorithms carefully find a small subset $S(t)$ of nodes around $u$ and $w$ whose PageRank values need to be updated. By contracting all other vertices $V \backslash S(t)$ to a single super node $s$, a small network $\bar{G}(t)$ is obtained. Then, these algorithms compute PageRank scores on the small network $\bar{G}(t)$ using static PageRank algorithms. The main disadvantage of these method is in accuracy. It depends largely on the choice of the subset $S(t)$. Although several methods for choosing $S(t)$ had been discussed in [13], but no theoretical guarantee was provided. Furthermore, even if the approximation error at time t is small, approximation errors can accumulate in long-term evolution [17].

- Monte Carlo based Algorithms

In order to explain the computational properties of PageRank, [16] shows that the general theory of Markov chains [18] can be applied to PageRank if the Web does not contain dangling nodes [4]. Under this hypothesis, [2] proposed and analyzed several Monte Carlo type methods for approximating PageRank, which allow to perform continuous update of the PageRank as the structure of the graph changes. These methods form the basis of Monte Carlo based algorithms for PageRank tracking. But they are very inefficient. Lately, [3] showed that a random walk segment needs to be adjusted only if it passes through the modified edge. However, denote $M_{t+1}$ as the actual number of random walk segments needed to be adjusted, they provided only a upper bound of $M_{t+1}$ in expectation for performance analysis as In Eq. (4), that can not be directly used in practice. This algorithm stores all random walk segments in a database, where each segment is stored at every node that it passes through, i.e., totally $\frac{nR}{\epsilon}$ random walk segments. For each node $u$, they also keep two counters: one, denoted by $W_u(t)$, keeping track of the number of random walk segments visiting $u$, and one, denoted by $outdeg_u(t)$, keeping track of the outdegree of $u$. Then, when an

edge $e(u, w)$ is inserted or removed, with probability $1 - (\frac{outdeg_u(t)-1}{outdeg_u(t)})^{W_u(t)}$ it adjusts all random walk segments passing through node $u$. Interestingly, when a graph is build from scratch with $k$ edges inserted, the time complexity is less than $\Omega(\frac{nR \ln k}{\epsilon})$ to keep PageRank scores updated all the time [3,15]. Another more efficient algorithm that stores no random walk segment was proposed in [14]. For each node $u$, this method keeps only a counter, denoted by $V_u(t)$, keeping track of the total times that all random walk segments visiting $u$. To adjust a random walk segment, we must first remove it, then simulate a new one. This method removes a previous random walk segment by simulating a new random walk and decrease $V_u(t)$ by 1 for every node $u$ it is passing through. Although the random walk segment to be removed and the new simulated one follow the same distribution, they are actually different segments. Hence, its way to adjust random walk segments also brings error.

## 4 Proposed Method

Here, we present a novel method for tracking PageRank on evolving networks. The proposed method is an adaptation of the Monte Carlo method. We first present the revisit probability model, which provides theoretical support for our algorithm. Then we propose our algorithm.

### 4.1 The Revisit Probability Model

Previous works did not consider the situation that a random walk may revisit an edge or a node. However, this situation is so common that we can not ignore it. Let $r_{uv}$ be the revisit probability of edge $e(u, v)$, we define it as following.

**Definition 1 (edge revisit probability).** *For a graph $G(t)$, $r_{uv}(t)$ is the probability that a random walk staring from node $u$ and picking $v$ as the next node (passing through edge $e(u, v)$) revisits the node $u$.*


(a) graph $G(t)$ (b) graph $G(t + 1)$

**Fig. 1.** Initial graph $G(t)$ and $G(t + 1)$ with edge $e(1, 4)$ inserted.

To get a better sense of $r_{uv}(t)$, we use Example 1 for explanation. It should be aware that $r_{uv}$ is determined only by the reset probability $\epsilon$ and the graph structure. Before describe Example 1, we provide the definitions of random walk path and random walk segment here.

**Definition 2 (random walk path).** *A random walk path is an unique and finite sequence of nodes denoted as* $\{u_1, u_2, u_3, u_4, ..., u_n\}$*, where there is an edge from* $u_{i-1}$ *to* $u_i$*.*

**Definition 3 (random walk segment).** *A random walk segment is an instance of a random walk path.*

*Example 1.* In this example, we calculate $r_{12}(t)$ on graph $G(t)$ in Fig. 1(a) with $\epsilon = 0.15$. We let a random walk terminate: with probability $\epsilon$ at each step; when it reaches a dangling node; when it revisits node ①. We list the probability of all random walk paths starting from edge $e(1, 2)$ in Table 1. Among all these random walk paths, only path 4 revisits node ①. Therefore, by Definition 1 we get $r_{12}(t) = 0.36125$, which is quite a high probability.

**Table 1.** The probability of all paths starting from $e(1, 2)$ in G(t).

| No. | Path $x$ | $\mathbb{P}(X = x)$ | |
|---|---|---|---|
| 1. | ① → ② | 0.15 | 0.15 |
| 2. | ① → ② → ④ | $0.85 \times \frac{1}{2}$ | 0.425 |
| 3. | ① → ② → ③ | $0.85 \times \frac{1}{2} \times 0.15$ | 0.06375 |
| 4. | ① → ② → ③ → ① | $0.85 \times \frac{1}{2} \times 0.85$ | 0.36125 |
| Sum. | | | 1 |

Then we define the probability that a random walk starting from node $u$ revisits the node $u$ at time $t$ as

$$R_u(t) = \sum_{v \in \boldsymbol{out}_u(t)} \frac{1-\epsilon}{outdeg_u(t)} r_{uv}(t), \tag{5}$$

where $\boldsymbol{out}_u(t)$ is the set of nodes that $u$ pointing out to. Since any random walk can terminate at node $u$ with probability $\epsilon$, we should multiply it by $1 - \epsilon$.

Assume $W_u(t)$ is the number of random walk segments passing though $u$ at time $t$, and $V_u(t)$ is the total times visited by those random walk segments. The relation between $V_u(t)$ and $W_u(t)$ can be express with $R_u(t)$ as the sum of a geometric series

$$V_u(t) = W_u(t)(1 + R_u^1(t) + R_u^2(t) + ...) = \frac{W_u(t)}{1 - R_u(t)}. \tag{6}$$

For a large and complex network, it is impossible to compute $R_u(t)$ directly as in Example 1. So, we use an estimation of $R_u(t)$ as

$$\widetilde{R}_u(t) = 1 - \frac{W_u(t)}{V_u(t)}. \tag{7}$$

We also find the following two observations when a graph changes, which make it possible for accurate PageRank tracking.

**Observation 1.** *When an edge $e(u, w)$ is modified at time $t + 1$, $r_{uv}$ remains unchanged for any node $v \in \boldsymbol{out}_u(t)$ and $v \neq w$ , where $\boldsymbol{out}_u(t)$ is the set of nodes that $u$ pointing out to.*

*Proof.* $r_{uv}$ is affected if and only if any random walk path starting from edge $e(u, v)$ visits the modified edge $e(u, w)$ before it revisits node $u$. However, if a path visits the modified edge $e(u, w)$, then it must first pass through node $u$. So we prove the proposition.

Similarly, we can prove that the following Observation 2 is also true.

**Observation 2.** *When an edge $e(u, w)$ is modified at time $t + 1$, walk count $W_u(t+1)$ remains unchanged but visit count $V_u(t+1)$ is affected, i.e., $W_u(t+1) = W_u(t)$ but $V_u(t + 1) \neq V_u(t)$.*

We already know that a random walk segment needs to be adjusted only if it visits the modified edge. Denote $M_{t+1}$ as the actual number of random walk segments that need to be adjusted. With the revisit probability model, the core problem of PageRank tracking can be defined as

**Definition 4 (core problem).** *Given a graph $G(t)$ and the modified edge $e(u, w)$ at time $t + 1$, compute the value of $M_{t+1}$ by using $W_u(t)$, $V_u(t)$ and $R_u(t)$.*

In the rest part of this section, we show how to solve this problem. We treat adding an edge and removing an edge separately, since they are different. When adding an edge $e(u, w)$ at time $t + 1$, no random walk segment really passes through the new edge $e(u, w)$ at time $t$, i.e., things have not happened yet. And when removing an edge $e(u, w)$ at time $t + 1$, some random walk segments did pass through the removing edge $e(u, w)$ at time $t$.

**Adding an Edge.** When adding an edge $e(u, w)$ to a graph $G(t)$, we get $outdeg_u(t + 1) = outdeg_u(t) + 1$. And from Eq. (5), we have

$$R_u(t + 1) = \frac{(1 - \epsilon)r_{uw} + outdeg_u(t)R_u(t)}{outdeg_u(t + 1)}. \tag{8}$$

We first prove the following proposition:

**Proposition 1.** *When adding an edge $e(u, w)$ to a graph $G(t)$ at time $t + 1$, $M_{t+1}$ is given by:*

$$M_{t+1} = \frac{W_u(t)}{outdeg_u(t + 1) - outdeg_u(t)R_u(t)}. \tag{9}$$

*Proof.* When $W_u(t)$ random walks set off from node $u$ at time $t+1$, $\frac{outdeg_u(t)}{outdeg_u(t+1)}$ of them pick previous $outdeg_u(t)$ edges of $u$ as the next step, and $\frac{1}{outdeg_u(t+1)}$ of them pick the new adding edge $e(u, w)$. Any random walk picking the new

adding edge $e(u, w)$ will need to be adjusted. If we let the this part of them just terminate there, then all random walks will visit the new adding edge $e(u, w)$ no more than once. After several steps, $R_u(t)$ of those random walks picking previous $outdeg_u(t)$ edges return to the node $u$. These random walks set off from node $u$ again. We repeat this process until no random walk return to the node $u$. Therefore, we could count the total times that all $W_u(t)$ random walks visit node $u$ as

$$V_{all} = W_u(t)(1 + \frac{outdeg_u(t)R_u(t)}{outdeg_u(t+1)} + ...) = \frac{W_u(t)}{1 - \frac{outdeg_u(t)R_u(t)}{outdeg_u(t+1)}}. \tag{10}$$

Since we let a random walk terminate when it visits the new adding edge $e(u, w)$, it is obvious that $M_{t+1} = \frac{V_{all}}{outdeg_u(t+1)}$. So we prove the proposition.

**Removing an Edge.** Removing an edge $e(u, w)$ from a graph $G(t)$ can be viewed as an inverse modification of adding an edge, so we have $outdeg_u(t) = outdeg_u(t+1) + 1$ and

$$R_u(t) = \frac{(1-\epsilon)r_{uw} + outdeg_u(t+1)R_u(t+1)}{outdeg_u(t)}. \tag{11}$$

Similarly to adding an edge, the following Proposition 2 also proves to be true.

**Proposition 2.** *When removing an edge $e(u, w)$ from a graph $G(t)$ at time $t+1$, $M_{t+1}$ is given by:*

$$M_{t+1} = \frac{W_u(t)}{outdeg_u(t) - outdeg_u(t+1)R_u(t+1)}. \tag{12}$$

However, we only know the estimation of $R_u(t)$ instead of $R_u(t+1)$. Remember that our method stores all $nR$ random walk segments. So when removing an edge $e(u, w)$ from a graph, $M_{t+1}$ is just the number of random walk segments passing through edge $e(u, w)$. This problem is also solved.

### 4.2 Algorithm for Tracking PageRank

For a graph with $n$ nodes, our algorithm maintains only $nR$ random walk segments ($R$ random walks starting from each node) in memory. Each random walk segment is assigned an unique id, called segment id. We save all these random walk segments in a hash-table, so that they can be accessed in $O(1)$ time. For each node $u$, our method also keeps a counter and a set: the counter, denoted by $V_u(t)$, keeping track of the total times that all random walk segments visiting $u$ at time $t$, the set, denoted by $S_u(t)$, keeping track of all random segments' id who pass through node $u$ at time $t$. Then the number of random walk segments passing through $u$, denoted by $W_u(t)$ is equal to size of $S_u(t)$, i.e., $W_u(t) = |S_u(t)|$. Every time we add or remove a random walk segment, $V_u(t)$ and $S_u(t)$ are automatically updated in $O(1/\epsilon)$ time as in Algorithm 1.

**Algorithm 1.** Segment Management

Let $S_u(t)$ be a set, $V_u(t)$ be a value kept for node $u$ and $sid$ be the segment id.

```
1: function AddSegment(sid, newSeg)
2:     for each u in newSeg do
3:         V_u(t) += 1
4:         Add sid to S_u(t)
5:     end for
6:     Add newSeg to hash-table with key sid
7: end function
8:
9: function RemoveSegment(sid)
10:    Get segment seg from hash-table with key sid
11:    for each u in seg do
12:        V_u(t) -= 1
13:        if sid in S_u(t) then
14:            remove sid from S_u(t)
15:        end if
16:    end for
17:    Remove seg from hash-table
18: end function
```

**Algorithm 2.** Adding or removing node

Let $u$ be the node added or removed.

```
1: function AddNode(u)
2:     for i = 0 → R do
3:         sid, newSeg ← simulate a random walk starting from w
4:         Call AddSegment(sid, newSeg)
5:     end for
6: end function
7:
8: function RemoveNode(u)
9:     for each sid whose segment starts from u do
10:        Call RemoveSegment(sid)
11:    end for
12: end function
```

The original static Monte Carlo method in [2] is used as an initial solution for our method. When adding an edge $e(u, w)$ to a graph $G(t)$ at time $t + 1$, for each random walk segment in $S_u(t)$, with probability $M_{t+1}/W_u(t)$ we redo it starting from the new edge $e(u, w)$. That means, we redo a random walk segment starting from where the first time it visits node $u$, and force it to pick node $w$ as the next node. When removing an edge $e(u, w)$ from a graph $G(t)$ at time $t + 1$, all random walk segments passing through edge $e(u, w)$ need to be adjust. For each random walk segment, we simply redo it starting from where the first time it visits node $u$.

Sometimes, when adding or removing an edge $e(u, w)$, nodes $u$ and $w$ are also added or removed. Remember that the original Monte Carlo method simulates

exactly $R$ random walks starting from each node in a graph. So, when a node $u$ is added to a graph, we also simulate $R$ random walks starting from $u$. And when a node $u$ is removed from a graph, we also remove all random walk segments starting from $u$. The proposed method is summarized in Algorithms 2 and 3.

**Algorithm 3.** Adding or removing edge

Let $e(u, w)$ be the edge modified at time $t + 1$.
**function** AddEdge($e(u, w)$)
  Compute $M_{t+1}$ as in Eq. (9)
  Add $e(u, w)$ to graph $G(t)$
  **for** each *sid* in $S_u(t)$ **do**
    **if** $random(0, 1) \leq \frac{M_{t+1}}{|S_u(t)|}$ **then**
      $newSeg \leftarrow$ redo segment starting from the new edge $e(u, w)$.
      Call RemoveSegment(*sid*)
      Call AddSegment($sid, newSeg$)
    **end if**
  **end for**
  **if** node $u$ is new added **then**
    Call AddNode(u)
  **end if**
  **if** node $w$ is new added **then**
    Call AddNode(w)
  **end if**
**end function**

**function** RemoveEdge($e(u, w)$)
  Remove $e(u, w)$ from graph $G(t)$
  **for** each *sid* whose segment passes through $e(u, w)$ **do**
    $newSeg \leftarrow$ redo segment starting from the first node $u$.
    Call RemoveSegment(*sid*)
    Call AddSegment($sid, newSeg$)
  **end for**
  **if** node $u$ is removed **then**
    Call RemoveNode($u$)
  **end if**
  **if** node $w$ is removed **then**
    Call RemoveNode($w$)
  **end if**
**end function**

## 5 Experiments

In this section, we conducted experiments with real-world dynamic networks. The overview of our experiments is as follows:

- Experimental settings (Sect. 5.1): The platform and datasets we used in our experiments are present here. We also introduced the comparison methods and parameter settings.
- Accuracy (Sect. 5.2): We evaluated the accuracy of the proposed algorithm. We first verified that the proposed method does not accumulate errors in long term evolution. Then we compared our algorithm to existing methods.
- Efficiency and scalability (Sect. 5.3): We first investigated the average update time for a single edge modification. We observed that the proposed algorithm is 1.3–30 times faster than state-of-the-art algorithms. Then we showed that the proposed algorithm is also suitable for large networks.

### 5.1 Experimental Settings

We conducted the experiments on a desktop computer with 8 GB of RAM and AMD CPU R5-2400G@ 3.60 GHz running Windows 10. 5 real-world dynamic graphs with time-stamps, obtained from http://snap.stanford.edu/data/ and http://konect.cc/networks/, were used in our experiments, which are listed in Table 2.

**Table 2.** Dynamic networks that we experimented with.

| Dataset | email-Enron | email-Eu-core | wiki-talk-ja | facebook-wosn | sx-askubuntu |
|---|---|---|---|---|---|
| Static nodes | 3.6K | 1K | 0 | 52K | 0 |
| Dynamic nodes | 75K | 0 | 397K | 11K | 159K |
| Static edges | 43K | 25.6K | 0 | 1.28M | 0 |
| Dynamic edges | 3.0M | 33.2K | 1.0M | 1.81M | 964K |

During our experiments, we compared our approach with the IMCPR algorithm proposed in [14], and the BahmaniPR algorithm proposed in [3]. The original static Monte Carlo based PageRank algorithm (MCPR) proposed in [2] was used as the ground-truth method. A detailed description of how these algorithms work could be found in Sects. 3 and 4.

There are two errors in the IMCPR method that make it perform very poor and unstable. Firstly, it computes $M_{t+1}$ by $V_u(t)/(outdeg_u(t)-1)$ when removing an edge, which will cause a divide by zero error if $outdeg_u(t) = 1$. Secondly, it adds or removes an edge before it simulates a random walk to remove a previous random walk segment. The order is wrong, which causes the random walk segment to be removed and the new simulated one do not follow the same distribution. We fixed these two errors and named the new version IMCPR2, which is the actual algorithm we used for comparison.

All algorithms were implemented in Python 2.7, and ran with parameters $\epsilon = 0.15$ and $R = 16$.

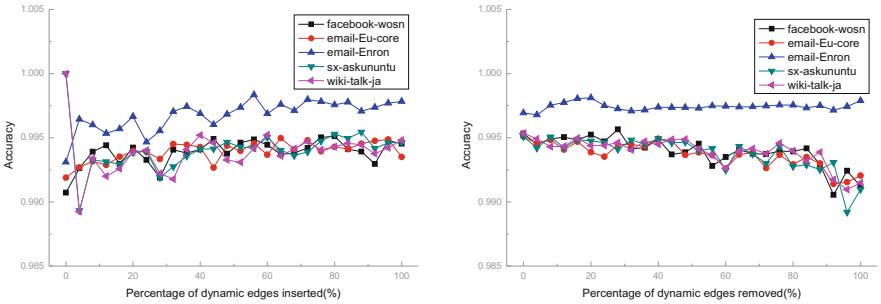
### 5.2 Accuracy

We first evaluated the accuracy of the proposed method. The accuracy is measured by cosine similarity defined as

$$Accuracy = \cos(\widetilde{\boldsymbol{\pi}}, \boldsymbol{\pi}),$$

where $\boldsymbol{\pi}$ is the "ground-truth" PageRank vector by MCPR and $\widetilde{\boldsymbol{\pi}}$ is the approximation by evaluated algorithms.

For adding edges, the initial network $G(0)$ was set as a network with all static edges, and an initial PageRank solution was also computed by the MCPR method. Then we inserted dynamic edges one-by-one in time order, which follows the preferential attachment model [1]. Similarly, for removing edges, we set the initial network $G(0)$ as the whole graph with both static edges and dynamic edges. We then sequentially removed dynamic edges from the current network in reverse time order. We traced the accuracy every time 4% percentages of dynamic edges were inserted or deleted. The results are shown in Fig. 2. These show that the proposed algorithm does not accumulate any errors in long-term evolution.

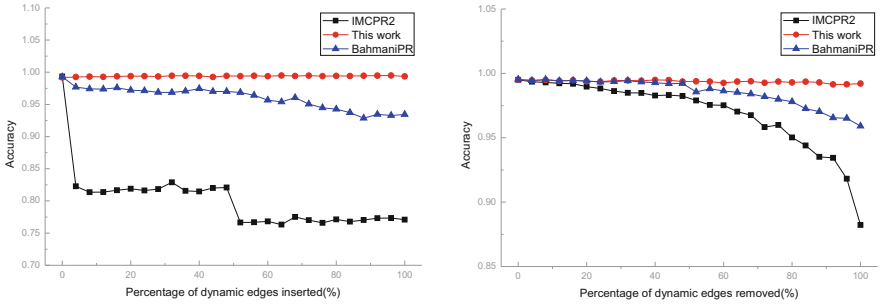



(a) adding edges. (b) removing edges.

**Fig. 2.** PageRank tracking accuracy of this work on 5 real-world networks.

We also plotted the comparison results on email-Eu-core network in Fig. 3. Results show that the proposed method performs best both in adding and deleting edges. And IMCPR2 accumulates errors more faster than BahmaniPR in long-term evolution. The reasons are as follows:

- Both IMCPR2 and BahmaniPR lack theory supports, so the estimation of $M_{t+1}$ or the probability they used for PageRank tacking are not accurate;
- IMCPR2 does not save any random walk segment, therefore its way to adjust random walk segments also brings error;
- IMCPR2 deals with an added or removed node u by setting $V_u(t) = R$ or 0, which is not correct.




(a) adding edges. (b) removing edges.

**Fig. 3.** Accuracy comparison of the algorithms on network email-Eu-core.

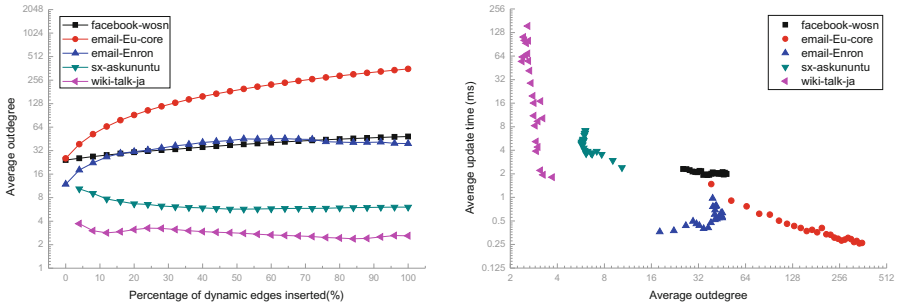
### 5.3 Efficiency and Scalability

Here, we investigated the efficiency and scalability of the proposed method. As in the previous accuracy evaluation, the network with only static edges was set as the initial network $G(0)$ for adding edges, and the whole graph with both static edges and dynamic edges was set as the initial network $G(0)$ for removing edges. We measured the average update time for a single edge insertion and deletion. The results are list in Table 3. These show that the proposed method is about 1.3 times faster than IMCPR2 algorithm and 30 times faster than BahmaniPR algorithm. BahmaniPR method is slow because it adjusts too many unnecessary random walk segments to update PageRank scores for a single edge modification. And since IMCPR2 method does not save any random walk segment, it has to simulate $2M_{t+1}$ random walks for each edge modification. However, our experiments show that simulating $M_{t+1}$ random walks is much slower than updating $M_{t+1}$ saved random walk segments in memory, especially for graphs with weighted edges.

**Table 3.** Average update time (ms) for inserting or deleting a single edge.

| Dataset | email-Enron | | email-Eu-core | | wiki-talk-ja | | facebook-wosn | | sx-askubuntu | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ins | del | ins | del | ins | del | ins | del | ins | del |
| This work | 0.57 | 0.46 | 0.41 | 0.29 | 42.0 | 36.4 | 2.06 | 1.87 | 4.83 | 3.17 |
| IMCPR2 | 0.89 | 0.65 | 0.73 | 0.64 | 65.8 | 49.2 | 3.43 | 2.56 | 8.01 | 5.19 |
| BahmaniPR | 20.6 | 17.4 | 16.5 | 13.2 | – | – | 50.3 | 46.2 | 74.1 | 59.8 |

We also verified that the proposed algorithm is scalable for large networks. For a graph with n nodes, the update time for a single edge modification is actually inversely proportional to the average outdegree $|\boldsymbol{E}|/n$, where $\boldsymbol{E}$ is the edge set. To verify this claim, we plotted the relation between the average update




(a) average outdegree (b) outdegree versus update time

**Fig. 4.** Scalability of the proposed method.

time and the average outdegree $|\boldsymbol{E}|/n$ in Fig. 4(b). We measured the average outdegree and the average update time every time 4% percentages of dynamic edges were inserted. Results on real-world datasets show that the average update time of 4 networks except email-Enron are inversely proportional to their average outdegree. And the email-Eu-core network, which does not contain any added or removed nodes, perfectly supports our claim. Figure 4(a) also shows that the average outdegree of a graph keeps stable or increases when it grows larger. Therefore our algorithm is also suitable for large networks.

## 6 Conclusions

In this paper, we proposed an efficient PageRank tracking algorithm on dynamic networks, which does not accumulate errors in long term evolution. The proposed algorithm is practically 1.3–30 times faster than state-of-the-art algorithms. We also presented a revisit probability model, which overcomes existing limitations of state-of-the-art Monte Carlo algorithms for tracking PageRank in dynamic networks. In future work, we hope to extend our method to Personalized PageRank [9,12,20] as well.

## References

1. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Rev. Mod. Phys. **74**, 47–97 (2002). https://doi.org/10.1103/RevModPhys.74.47. https://link.aps.org/doi/10.1103/RevModPhys.74.47
2. Avrachenkov, K., Litvak, N., Nemirovsky, D., Osipova, N.: Monte Carlo methods in PageRank computation: when one iteration is sufficient. SIAM J. Numer. Anal. **45**(2), 890–904 (2007). https://doi.org/10.1137/050643799
3. Bahmani, B., Chowdhury, A., Goel, A.: Fast incremental and personalized PageRank. Very Large Data Bases **4**(3), 173–184 (2010). https://doi.org/10.14778/1929861.1929864

4. Bianchini, M., Gori, M., Scarselli, F.: Inside PageRank. ACM Trans. Internet Technol. **5**(1), 92–128 (2005). https://doi.org/10.1145/1052934.1052938
5. Bonacich, P.: Power and centrality: a family of measures. Am. J. Sociol. **92**(5), 1170–1182 (1987). https://doi.org/10.1086/228631
6. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Int. World Wide Web Conf. **30**, 107–117 (1998). https://doi.org/10.1016/S0169-7552(98)00110-X
7. Chien, S., Dwork, C., Kumar, R., Simon, D.R., Sivakumar, D.: Link evolution: analysis and algorithms. Internet Math. **1**(3), 277–304 (2004). https://doi.org/10.1080/15427951.2004.10129090
8. Desikan, P.K., Pathak, N., Srivastava, J., Kumar, V.: Incremental Page Rank computation on evolving graphs. In: International World Wide Web Conferences, pp. 1094–1095 (2005). https://doi.org/10.1145/1062745.1062885
9. Guo, W., Li, Y., Sha, M., Tan, K.L.: Parallel personalized PageRank on dynamic graphs. Proc. VLDB Endow. **11**(1), 93–106 (2017). https://doi.org/10.14778/3151113.3151121
10. Kamvar, S.D., Haveliwala, T.H., Manning, C.D., Golub, G.H.: Exploiting the block structure of the web for computing PageRank. Stanford University Technical Report (2003)
11. Katz, L.: A new status index derived from sociometric analysis. Psychometrika **18**(1), 39–43 (1953). https://doi.org/10.1007/BF02289026
12. Kloumann, I.M., Ugander, J., Kleinberg, J.: Block models and personalized PageRank. Proc. Natl. Acad. Sci. U. S. A. **114**(1), 33 (2017). https://doi.org/10.1073/pnas.1611275114
13. Langville, A.N., Meyer, C.D.: Updating markov chains with an eye on google's PageRank. SIAM J. Matrix Anal. Appl. **27**(4), 968–987 (2006). https://doi.org/10.1137/040619028
14. Liao, Q., Jiang, S.S., Yu, M., Yang, Y., Li, T.: Monte Carlo based incremental PageRank on evolving graphs. In: Kim, J., Shim, K., Cao, L., Lee, J.-G., Lin, X., Moon, Y.-S. (eds.) PAKDD 2017. LNCS (LNAI), vol. 10234, pp. 356–367. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57454-7_28
15. Lofgren, P.: On the complexity of the Monte Carlo method for incremental PageRank. Inf. Process. Lett. **114**(3), 104–106 (2014). https://doi.org/10.1016/j.ipl.2013.11.006
16. Ng, A.Y., Zheng, A.X., Jordan, M.I.: Stable algorithms for link analysis. In: International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 258–266 (2001). https://doi.org/10.1145/383952.384003
17. Ohsaka, N., Maehara, T., Kawarabayashi, K.I.: Efficient PageRank tracking in evolving networks. In: ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 875–884 (2015). https://doi.org/10.1145/2783258.2783297
18. Seneta, E.: Non-negative matrices and Markov chains. Popul. Stud. J. Demogr. **37**(1), 476 (1981). https://doi.org/10.1007/0-387-32792-4
19. Tong, H., Papadimitriou, S., Yu, P.S., Faloutsos, C.: Proximity tracking on time-evolving bipartite graphs. In: SIAM International Conference on Data Mining, SDM 2008, Atlanta, Georgia, USA, 24–26 April 2008, pp. 704–715 (2008). https://doi.org/10.1137/1.9781611972788
20. Zhang, H., Lofgren, P., Goel, A.: Approximate personalized PageRank on dynamic graphs. In: ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1315–1324 (2016). https://doi.org/10.1145/2939672.2939804