Personalized Ranking in Dynamic Graphs Using Nonbacktracking Walks[*]

Eisha Nathan[1], Geoffrey Sanders[1], and Van Emden Henson[1]

Lawrence Livermore National Laboratory, Livermore CA 94550
{nathan4,sanders29,henson5}@llnl.gov

Abstract. Centrality has long been studied as a method of identifying node importance in networks. In this paper we study a variant of several walk-based centrality metrics based on the notion of a nonbacktracking walk, where the pattern $i \to j \to i$ is forbidden in the walk. Specifically, we focus our analysis on dynamic graphs, where the underlying data stream the network is drawn from is constantly changing. Efficient algorithms for calculating nonbactracking walk centrality scores in static and dynamic graphs are provided and experiments on graphs with several million vertices and edges are conducted. For the static algorithm, comparisons to a traditional linear algebraic method of calculating scores show that our algorithm produces scores of high accuracy within a theoretically guaranteed bound. Comparisons of our dynamic algorithm to the static show speedups of several orders of magnitude as well as a significant reduction in space required.

Keywords: Non-backtracking walks · Dynamic graphs · Centrality.

1 Introduction

Calculating node rankings is a commonly studied problem in graph analysis, typically done to identify the "most important" vertices in a network. Several centrality metrics are calculated by quantifying traversals around a network, or by counting walks, where a walk in a graph is a sequence of vertices that allows for both vertices and edges to repeat. More recently, several authors have presented the notion that all walks are not created equally and more importance ought to be given to walks that do not *backtrack* on themselves (walks that visit a particular vertex i, visit its neighbor j, then immediately backtrack to vertex i) [1]. In the case of networks modeling disease spread, a walk that backtracks upon itself provides little to no useful information about the spread of disease. Backtracking walks in information diffusion networks do not allow the user to glean any new information. Furthermore, [2, 3] noted that localization effects can be avoided by studying these nonbacktracking walks. In this paper, we study

[*]This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 with release number LLNL-CONF-765259.

nonbacktracking walks and the associated centrality scores and propose a new algorithm for computing them. We additionally extend our analysis to dynamic graphs, where data is allowed to change over time, creating new relationships in the network. When studying analytics on dynamic graphs, it is important to have measures that can update efficiently given just the changes made to the graph from the previous timestep. This avoids a full recomputation every time the underlying graph is changed and avoids unnecessary computation time. The contributions of this paper can thus be summarized as:

- A new algorithm for approximating personalized nonbacktracking centrality scores in **static** graphs
- Theoretical and empirical proof that our static algorithm produces approximate values close to exact nonbacktracking centrality scores
- A new algorithm for approximating nonbacktracking centrality scores in **dynamic** graphs
- Evidence that our dynamic algorithm is more efficient than a static recomputation for real datasets

The rest of the paper is organized as follows. Section 2 presents the necessary notation used to understand the problem and a brief overview of some related work. Section 3 presents both the static and dynamic algorithms and Section 4 presents our results. In Section 5 we conclude.

2 Background

2.1 Terminology

Let $G = (V, E)$ be a graph, where V is the set of n vertices and E the set of m edges. A is the $n \times n$ adjacency matrix of a graph where $A_{ij} = 1$ if there is an edge between vertices i and j in the graph, and 0 otherwise. Although all the work presented in this paper can be applied to both undirected and directed graphs, here we focus on undirected graphs so $\forall e = (i, j) \in E$, $A_{ij} = A_{ji} = 1$. Dynamic graphs represent data that is changing over time and can be modeled as snapshots of the current state of the data at different points in time, or a sequence of static graphs. Dynamic graphs can be thought of as graph data that has timestamps associated with it. Denote the current snapshot at time t of the dynamic graph G and its corresponding adjacency matrix A as $G_t = (V_t, E_t)$ and A_t respectively. The difference in subsequent snapshots of the graph at different points in time t and $t+1$ can be written as $\Delta A = A_{t+1} - A_t$, where ΔA represents the change to the graph at time t. If an edge (i, j) is inserted into the graph at time t, then $\Delta A_{ij} = 1$. A walk of length k in a graph is a series of connected vertices $v_1, v_2, \cdots, v_k$, where vertices are allowed to repeat. Powers of the adjacency matrix are used to count walks of different lengths where A^k_{ij} is the number of walks of length k from vertex i to j [4]. A nonbacktracking walk (NBTW) is defined as a walk which does not backtrack upon itself, meaning it contains no vertex sequences of the form $i \to j \to i$. Let $N(i)$ denote the set of neighbors of vertex i. For a particular NBTW w that ends with the sequence of vertices $\cdots, j, i$, let $\widetilde{N}_j(i) = N(i)\backslash j$, meaning the set of neighbors of vertex i without the vertex the NBTW w came from (vertex j).

2.2 Related Work

Several walk-based centrality metrics are calculated as functions of the adjacency matrix [5]. Katz Centrality, for example, weights walks starting at each vertex in the graph of different lengths by increasing powers of some parameter α, where longer walks are given less importance [6]. The parameter α must fall somewhere in the range $(0, 1/\|A\|_2)$, where $\|A\|_2$ is the 2-norm of A. As α reaches its upper limit, however, the centrality scores correspond to eigenvector centrality [7]. A subset of these walk-based centrality metrics and their generalized equation is given in Table 1.

Table 1: Several walk-based centralities as functions of the adjacency matrix

Centrality Metric	**Generalized Equation**
Katz Centrality [6] PageRank [8]	$\sum_{k=0}^{\infty} \alpha^k A^k$
Eigenvector centrality [7]	$A\mathbf{x} = \lambda \mathbf{x}$
Exponential centrality [9] Subgraph centrality [10] Total communicability [11]	$\sum_{k=0}^{\infty} \frac{A^k}{k!}$

The similarity amongst all these walk-based centrality metrics stems from the fact that they weight all walks of the same length equally. For example, a walk of length 4 between vertices $0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0$ is given the same weight as a walk of length 4 between vertices $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. Therefore, a new measure of centrality was proposed in [1], based on the concept of a *nonbacktracking walk*. Nonbacktracking walk centrality scores are computed by counting NBTWs in graphs and weighting longer ones by successive powers of some parameter $\alpha \in (0, 1)$. In this paper we calculate personalized centrality scores (w.r.t. seed vertices of interest) using NBTWs by counting NBTWs originating at some seed vertex and ending at all other vertices in the graph.

When NBTW-centrality was first introduced in [1], the authors presented a linear algebraic formulation for calculating the centrality scores based off of a *deformed graph Laplacian*. This was later extended to analysis on directed networks in [12] and in [2] a nonbacktracking variant of eigenvector centrality was introduced. The infinite sum in Theorem 1 converges to the exact solution for the scores $\mathbf{x}^*$, and if the vector $\mathbf{1}$ is replaced with $\mathbf{e}_i$ we obtain the personalized scores w.r.t. a seed vertex i instead of the global scores. Here, D is the associated diagonal degree matrix of the adjacency matrix A.

Theorem 1. *For $P_k = AP_{k-1} + (I - D)P_{k-2}$, where D is the diagonal degree matrix of A, $P_0 = I$, $P_1 = A$, and $P_2 = A^2 + (I - D)$, $\mathbf{x}^* = \sum_{k=1}^{\infty} \alpha^k P_k \mathbf{1}$ [1].*

This sum converges to the linear system in Equation 1.

$$(I - \alpha A + \alpha^2(D - I))\mathbf{x}^* = (1 - \alpha^2)\mathbf{e}_i \tag{1}$$

However, for large graphs, this linear system is computationally intensive to solve [13] and for personalized scores, the scores of vertices far away from the seed are often negligible. Therefore, it is desirable to have an alternate method to calculate these centrality scores and in this work we present one such alternate algorithm by directly tabulating walks up to a certain length. Since walks (and NBTWs) in graphs can be infinitely long, if we are counting walks manually (without using linear algebra), we can approximate the corresponding centrality metric by counting walks up to a certain length. We use the notation developed in Theorem 1 as the foundation for our algorithm. Furthermore, for dynamic updates, the linear algebraic approach of solving a linear system has several disadvantages. The linear algebraic approach will take at least several matrix-vector multiplications (on the order of $\mathcal{O}(m)$ to converge to a new solution every time the graph is updated). If the update to the graph is only a few edge insertions, this much computation is unnecessary and is avoided by our approach. Therefore, temporal fidelity is limited.

Apart from ranking, nonbacktracking walks have been studied in the context of community detection as well. Community detection is the task of identifying groups of vertices more closely related to each other than to the rest of the network [14]. In [15], a nonbacktracking variant of spectral clustering was proposed. By using the nonbacktracking matrix B, a $2m \times 2m$ matrix with entries $B_{(u\to v),(w\to x)} = 1$ if $v = w$ and $u \neq x$ and 0 otherwise, the authors show that performance of spectral algorithms in sparse networks fares better compared to using other commonly used linear operators. They show that the spectrum of this operator maintains a stronger separation between the eigenvalues relevant to showing community structure and the rest of the eigenvalues than other matrices that are more frequently used. Results are optimal for stochastic block model graphs, synthetic networks that have associated ground truth for community structure.

3 Algorithms

3.1 Approximation Theory

The logic behind our static algorithm is as follows: if we are interested in NBTW-centrality scores w.r.t. seed vertex *seed*, we count all NBTWs originating from *seed* up to some maximum walk length k. Tabulating walks in this manner inherently introduces error into the final solution since we do not count walks up to infinite lengths in the network. Let $\mathbf{x}^*$ be the solution to the linear system (the exact NBTW-centrality scores) and $\mathbf{x}_k$ be the approximation from our algorithm by counting up to length k NBTWs. We can bound the error between $\mathbf{x}^*$ and $\mathbf{x}_k$ as in Theorem 2. Using the notation from Theorem 1, our approximation $\mathbf{x}_k$ can mathematically be written as $\mathbf{x}_k = \sum_{r=1}^{k} \alpha^r P_r \mathbf{1}$.

Theorem 2. *The error between the exact solution and the* k*th approximation can be bounded by* $\|\mathbf{x}^* - \mathbf{x}_k\|_2 \leq \epsilon_k$ *where* $\epsilon_k = \frac{(\alpha\phi\|A\|)^{k+1}}{1-(\alpha\phi\|A\|)}$ *for* $(\alpha\phi\|A\|) < 1$*, where* $\phi = \frac{1+\sqrt{5}}{2}$.

Proof. We first bound the norm of P_k for any k:

– Let $\rho_k = \|P_k\|_2$
– Using the recursive formula for P_k from Theorem 1, we have $\rho_k \leq \|A\|\rho_{k-1} + \|I - D\|\rho_{k-2}$
– This leads to a closed-form solution for ρ_k:
  - Let $\rho_k = a^k$ where a is the root to characteristic polynomial $a^2 - \|A\|a - \|I - D\| = 0$
  - $a = \frac{1}{2}(\|A\| \pm \sqrt{\|A\|^2 + 4\|D - I\|} \leq \frac{1+\sqrt{5}}{2}\|A\|$

  $\Rightarrow$ $\rho_k = (\phi\|A\|)^k$

$$\begin{aligned}
\|\mathbf{x}^* - \mathbf{x}_k\|_2 &= \|\sum_{r=1}^{\infty}\alpha^r P_r - \sum_{r=1}^{k}\alpha^r P_r\|_2 = \|\sum_{r=k+1}^{\infty}\alpha^r P_r\|_2 \\
&\leq \sum_{r=k+1}^{\infty}\alpha^r\|P_r\|_2 = \sum_{r=k+1}^{\infty}\alpha^r(\phi\|A\|)^r = \sum_{r=0}^{\infty}(\alpha\phi\|A\|)^{r+k+1} \\
&\leq (\alpha\phi\|A\|)^{k+1}\sum_{r=0}^{\infty}(\alpha\phi\|A\|)^r = \frac{(\alpha\phi\|A\|)^{k+1}}{1-(\alpha\phi\|A\|)} =: \epsilon_k
\end{aligned}$$

Results shown in Section 4 demonstrate that our method produces centrality scores at least within ϵ_k of the exact solution as we count up to length k NBTWs.

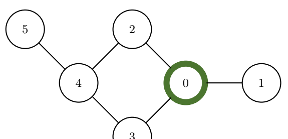
3.2 Static Algorithm

For a graph with n vertices, we maintain an $n \times k$ array *walks* where *walks[i][j]* represents the number of nonbacktracking walks from *seed* to vertex i of length j. The effect of a walk from *seed* to one of its direct neighbors can be propagated throughout the network, where we only advance the walk to a vertex if we don't backtrack. Since walks are required to be nonbacktracking, at step r we need to keep track of the vertex that was visited at step $r - 1$.

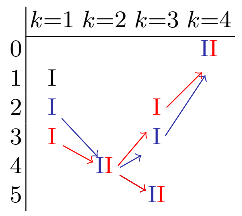
We use a priority queue [16] to keep track how many NBTWs of different lengths exist in the network at any given point. The priority queue is filled with 4-tuple elements of the type $(prev, curr, k, num\_walks)$, where an element u in the priority queue means we are currently processing $u.num\_walks$ of length $u.k$ ending at $u.curr$ that came from $u.prev$. The queue is prioritized by k, meaning elements with higher values of k are processed first, searching in a manner consistent with depth-first search, starting from the seed set. If the queue is not prioritized by k and the algorithm is allowed to search in a breadth-first manner (starting from the seed, then adding all immediate neighbors of the seed, then neighbors one step out), the size of the queue would grow exponentially and become impractical to use for very large graphs memory-wise.

Figure 1 gives an example of our static algorithm on a toy network. The example graph is shown in Figure 1a with a seed vertex 0 outlined in green. Figure 1c shows the progression of the priority queue as we process elements.

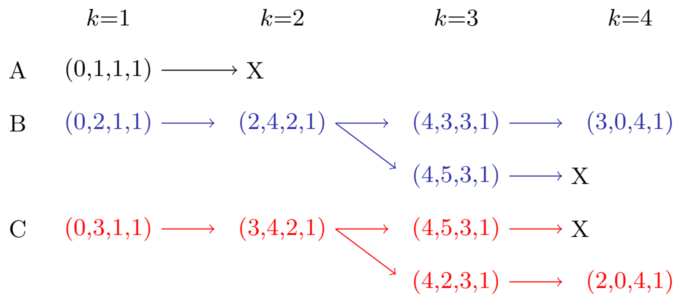
Walks that are terminated (either due to reaching the maximum length or to the lack of neighbors) are depicted by an 'X.' Since the seed vertex 0 has three distinct neighbors (vertices 1, 2, and 3) we initialize the priority queue with their respective elements (the three elements under the k=1 heading). Each of the elements is processed: since vertex 1 has no neighbors that don't involve backtracking, the element (0,1,1,1) is removed from the priority queue and no new elements are added (depicted in row A). The element (0,2,1,1) at the start of row B indicates we have 1 NBTW from vertex 0 to 2. We follow the progression of this NBTW through the priority queue shown by the blue elements. The corresponding NBTW counts is also shown in blue in Figure 1b. A similar progression is shown for the walk starting from vertex 0 to 3 in the red elements (starting at row C in the priority queue). Note that elements are processed in priority order according to the value of k, to ensure a depth first traversal (row A first, then row B, then row C).



(a) Static graph with seed vertex 0 in green outline.



(b) Walks array.



(c) Priority queue.

Fig. 1: Example of STATIC_NBTW. Propagation of different walks is shown in different colors. For a seed vertex of 0, we propagate walks from neighbors vertex 1, 2, and 3 throughout the network.

Algorithm 1 gives the overall static algorithm for counting all the NBTWs up to length *k_max* in the network originating at the *seed* vertex. By the definition of a NBTW, the only vertices that will have a NBTW of length 1 from *seed* are the its direct neighbors. Line 2 initializes the priority queue with the seed's neighbors obtained in Line 3. For each of the *seed*'s neighbors *nbr*, there is one NBTW of length 1 from the *seed* to *nbr* (Lines 4-6). For each of these NBTWs, their effect is then propagated throughout the rest of the network. A new element is created to be inserted into the priority queue in Line 7, indicating that we have one walk of length 1 from *seed* to *nbr*.

Algorithm 1 Count NBTWs up to length *k_max* from a seed vertex *seed* in a static graph.

```
1: procedure STATIC_NBTW($seed, k\_max$)
2:     PriorityQueue $*pq$ = NEW PriorityQueue
3:     $Nbrs = N(seed)$
4:     for $nbr$ in $Nbrs$ do
5:         $num\_walks = 1$
6:         $walks[nbr][1] = num\_walks$
7:         $new\_elt = (seed, nbr, 1, num\_walks)$
8:         $pq \rightarrow$INSERT($new\_elt$)
9:     $walks$ = EVALUATE_PRIORITY_QUEUE($pq$,$k_{max}$)
10: return $walks$
```

The main computation occurs in Algorithm 2, where we iterate through the priority queue processing each element and counting NBTWs. For each element in the priority queue (Line 2), we update the number of NBTWs possible (Line 4). If we have not reached the maximum length of NBTWs we are counting (Line 5), we examine the set of neighbors (Line 6) of the current vertex associated with the element *elt* of the priority queue we are processing. For each vertex *nbr* in this neighbor set we add a new element to the priority queue indicating we have *elt.num_walks* new NBTWs of length $(elt.k + 1)$ ending in the sequence $\cdots, elt.prev, elt.curr, nbr$ (Line 8).

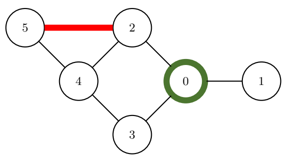
Algorithm 2 Process every element in the priority queue.

```
1: procedure EVALUATE_PRIORITY_QUEUE($pq, k_{max}$)
2:     while !$pq \rightarrow$IS_EMPTY() do
3:         $elt = pq \rightarrow$POP()
4:         $walks[elt.curr][elt.k] += elt.num\_walks$
5:         if $elt.k + 1 < k_{max}$ then
6:             $S = N(elt.curr) \backslash elt.prev$
7:             for $nbr \in Nbrs$ do
8:                 $new\_elt = (elt.curr, nbr, elt.k + 1, elt.num\_walks)$
9:                 $pq \rightarrow$INSERT($new\_elt$)
       return $walks$
```

3.3 Dynamic Algorithm

For dynamic graphs, a naive implementation to obtain updated NBTW counts after changes to the graph occur would recompute from scratch the number of NBTWs from *seed*. However, as the graph grows larger, this naive static recomputation becomes increasingly computationally intensive. By exploiting the locality of edge insertions we can develop a more efficient dynamic algorithm that only updates NBTW counts relevant to new edges inserted into the graph. We consider the case of inserting a single edge $e = (src, dest)$.

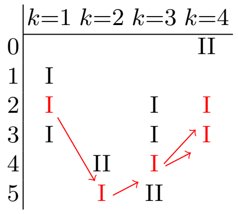
Figure 2 gives an example of our dynamic algorithm using the same toy network as earlier. Consider the effect of adding a single edge $e = (2, 5)$ (shown in red in Figure 2a). Here we do not show the priority queue at each stage, but rather the effect of updating NBTW counts in the *walks* array. Figure 2b gives the initial NBTW counts for the network before adding edge e. After inserting the edge between vertices 2 and 5, there are three NBTWs and their counts to update: 1) the NBTW of length 1 starting at vertex 2, 2) the NBTW of length 3 starting at vertex 2, and 3) two NBTWs of length 3 starting at vertex 5. These edge propagations are given in Figures 2c, 2d, and 2e respectively.



(a) Graph with seed vertex 0 and newly added edge $e = (2, 5)$ in red.

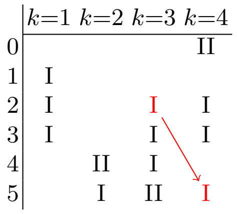
	k=1	k=2	k=3	k=4
0				II
1	I			
2	I		I	
3	I		I	
4		II		
5			II	

(b) Initial walks array before adding edge e.



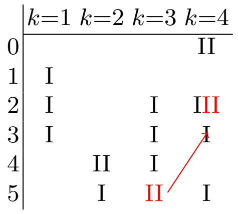
	k=1	k=2	k=3	k=4
0				II
1	I			
2	I		I	I
3	I		I	I
4		II	I	
5		I	II	

(c) Walks array after propagating NBTW of length k=1 ending at vertex 2.



	k=1	k=2	k=3	k=4
0				II
1	I			
2	I		I	I
3	I		I	I
4		II	I	
5		I	II	I

(d) Walks array after propagating NBTW of length k=3 ending at vertex 2.



	k=1	k=2	k=3	k=4
0				II
1	I			
2	I		I	IIII
3	I		I	I
4		II	I	
5		I	II	I

(e) Walks array after propagating NBTWs of length k=3 ending at vertex 5.

Fig. 2: Example of DYNAMIC_NBTW. After adding edge e between vertices 2 and 5 we show the steps of the dynamic algorithm to update NBTW counts taking into consideration the new edge.

Our dynamic algorithm is given in DYNAMIC_NBTW in Algorithm 3. All current NBTWs ending in *src* need to be updated since we can now visit *dest* from *src* by traversing the newly added edge. To identify which NBTWs need to be examined, we first find all the values of k where $walks[src][k]$ is nonzero in Line 3 (obtained in the array k_src). If $walks[src][k] > 0$, then there are a nonzero number of NBTWs of length k that end in *src* and we need to propagate these using the newly added edge e. The corresponding element is added to the priority queue in Line 6. This same procedure is repeated for the *dest* vertex in Lines

7-10. Lines 11-14 take care of the edge case when either *src* or *dest* is the seed vertex. In this case we need to perform a full propagation from the start similar to the static algorithm. Since we are not recalculating all the counts of NBTWs for all the vertices from *seed*, and are only examining the effect of a single edge, we only add elements to the priority queue corresponding to walks that use the newly added edge. Therefore, this dynamic approach will be significantly faster than a naive static recomputation every time the graph is changed and we see this in Section 4.

Algorithm 3 Dynamic algorithm for calculating NBTW centrality scores.

```
1: procedure DYNAMIC_NBTW($e = (src, dest)$)
2:     PriorityQueue $*pq$ = NEW PriorityQueue
3:     $k\_src = walks[src].nonzero$
4:     for $k$ in $k\_src$ do
5:         $num\_walks = walks[src][k]$
6:         $pq \rightarrow$INSERT($(src, dest, k+1, num\_walks)$)
7:     $k\_dest = walks[dest].nonzero$
8:     for $k$ in $k\_dest$ do
9:         $num\_walks = walks[dest][k]$
10:        $pq \rightarrow$INSERT($(dest, src, k+1, num\_walks)$)
11:    if $seed$ is $src$ then
12:        $pq \rightarrow$INSERT($(src, dest, 1, 1)$)
13:    if $seed$ is $dest$ then
14:        $pq \rightarrow$INSERT($(dest, src, 1, 1)$)
15:    $walks$ = EVALUATE_PRIORITY_QUEUE($pq$,$k_{max}$)
16: return $walks$
```

Both STATIC_NBTW and DYNAMIC_NBTW return an $n \times k$ array *walks* that can then be used to calculate the centrality scores. This procedure is given in Algorithm 4 where we obtain the centrality value for vertex i by weighting NBTWs of different lengths by successive powers of some user-chosen parameter $\alpha \in (0, 1)$.

Algorithm 4 Calculate NBTW-centrality scores from walk counts.

```
1: procedure CALCULATE_SCORES($walks$, $\alpha$)
2:     $\mathbf{x} = n \times 1$ array initialized to 0
3:     for $i = 1 : n$ do
4:         for $j = 1 : k$ do
5:             $\mathbf{x}[i]$ += $\alpha^j \cdot walks[i][k]$
       return $\mathbf{x}$
```

4 Results

We evaluate STATIC_NBTW and DYNAMIC_NBTW on five real-world graphs drawn from the KONECT collection [17]. Graph information is given in Table 2. For all results, five vertices from each graph are chosen randomly as seed vertices and results shown are averaged over these five seeds. For the dynamic experiments, we use temporal datasets to simulate dynamic graphs, meaning the edges already have associated timestamps. For our dynamic algorithm, we initialize the graph with half the edges and then insert the remaining edges in different batch sizes in timestamped order. We test batch sizes of 1, 10, 100, and 1000. A batch size of b means at each time point we insert b edges and run both the dynamic and static algorithms for comparison purposes. As previously discussed, many real graphs are small-world networks [18], meaning the graph diameter is on the order of $\mathcal{O}(\log(n))$. We set $k = \lceil \log(n) \rceil$, so by counting walks up to length $\approx \log(n)$, we can reach most vertices in the graph. The code was implemented in C++.

Table 2: Real graphs used in experiments.

Graph	$\|V\|$	$\|E\|$	Graph	$\|V\|$	$\|E\|$
karate	34	78	digg	279,630	1,731,653
lesmis	77	254	wiki-french	1,420,367	4,641,928
copperfield	112	425	wiki-english	2,987,535	24,981,163
slashdot	50,835	140,451	youtube	3,223,585	9,375,374

4.1 Static Algorithm Results

For our static algorithm we present comparisons to a conventional linear algebraic method of solving the system in Equation 1 discussed in Section 2. The goal here is to ensure our algorithm returns similar quality scores to a traditional linear algebraic computation of centrality scores. We measure error as the 2-norm difference between the two vectors as $error = \|\mathbf{x}^* - \mathbf{x}_k\|_2$. Results are shown for the three smallest graphs (COPPERFIELD, KARATE, LESMIS) to ensure we can solve the linear system accurately to present accurate comparisons.

Figure 3 plots the error (on the y-axis) for different values of k (on the x-axis) for the three smallest graphs. Each color corresponds to a specific graph and the dotted lines indicate the theoretically guaranteed error ϵ_k from Theorem 2 and the solid lines with square markers plot the obtained error from our approximation algorithm. The first trend to note is the most intuitive: as we include counts of longer NBTWs in the calculations of the scores, the error between our approximation and the exact scores decreases. Furthermore, our actual obtained error is always below that of the theoretically guaranteed error (usually by several orders of magnitude), showing that our approximation algorithm produces good quality scores compared to the exact linear algebraic scores.

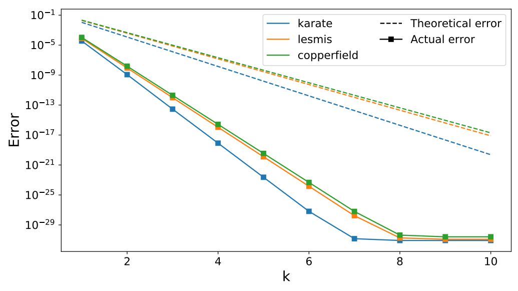


Fig. 3: Theoretical and actual error between exact NBTW-centrality scores $\mathbf{x}^*$ and our approximation $\mathbf{x}_k$.

4.2 Dynamic Algorithm Results

Our dynamic algorithm produces the same NBTW counts as our static algorithm (and therefore, the same scores), so we only examine the performance of our dynamic algorithm w.r.t. speedup in execution time compared to the static algorithm. Let T_S be the time taken by our static algorithm to compute the NBTW-based centrality scores for a particular graph and T_D be the time taken by our dynamic algorithm. We calculate the speedup in time as $speedup = \frac{T_S}{T_D}$. Higher values of the speedup indicate our dynamic algorithm has significant performance improvement compared to our static.

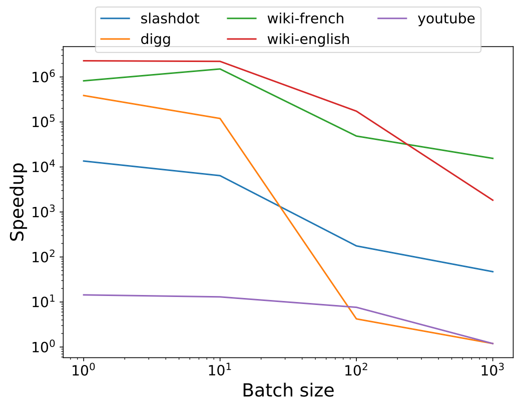


Fig. 4: Speedup versus batch size for real graphs.

Figure 4 plots the speedups for the five largest real graphs (on the y-axis) versus the batch size (on the x-axis). In all cases even the minimum speedup obtained is above 1×. We see the greatest speedup for smaller batch sizes of 1 and 10, indicating that our method is most beneficial for low latency applications with small number of data changes. The average speedup obtained decreases for larger batch sizes. This is due to the fact that as the batch size grows larger, the amount of time needed to process the updates grows because all endpoints of all newly added edges must be taken into account. Essentially, new NBTWs must

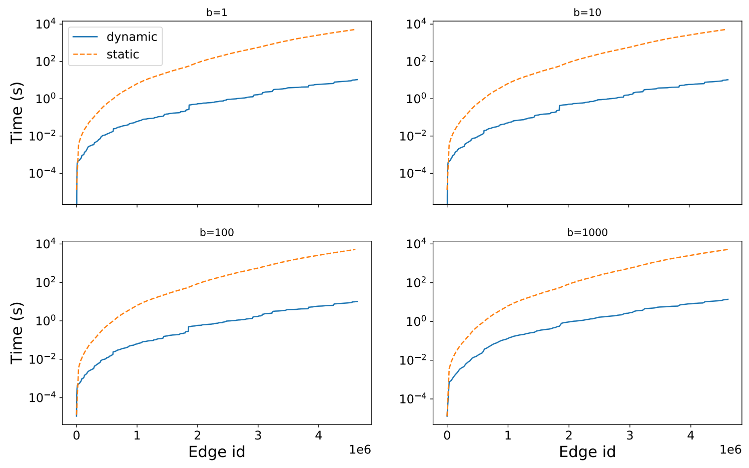


Fig. 5: Speedup aggregated over time for SLASHDOT graph.

be propagated from all the endpoints of the newly added edges. However, our dynamic algorithm still on average is able to obtain several orders of magnitude in speedup over the static recomputation. In very large graphs of millions of vertices where a static recomputation is computationally infeasible given edge updates to a graph, our dynamic algorithm offers significant savings because it just targets a localized portion of the graph where the edge has been added. Additionally, in applications where the entire graph is not able to fit in memory, our dynamic algorithm only needs to access portions of the graph directly affected by edge updates. However, since the overall trend shows decreasing speedup as the batch size is increased, this tells us that there is a batch size large enough at which it is computationally more efficient to recompute from scratch using the static algorithm.

Figure 5 plots the speedup over time for the WIKI-FRENCH graph. As edges are inserted into the graph and both algorithms are run, we plot the aggregated time taken in seconds for the dynamic (solid blue line) and the static (dotted orange line) algorithms. The y-value plotted at any given edge id m is the time taken by either the static or dynamic to process all edges up to and including edge m. Note that the y-axis is on a log scale. For this experiment, instead of starting with half the number of edges in the graph, we start with an empty graph for both the static and dynamic algorithms. Although the times taken by both algorithm are initially the same, over time we see that the time taken by our dynamic algorithm is several orders of magniture lower than the time taken by our static algorithm. While there is a spike in the time taken both algorithms initially, the aggregated times for the dynamic algorithm increase linearly (indicating processing each batch of edges at each time point takes more or less the same time), unlike those from the static algorithm. This indicates that our method of only examining places in the graph that are directly affected by the edge updates results in a highly efficient computation of the NBTW-based centrality scores.

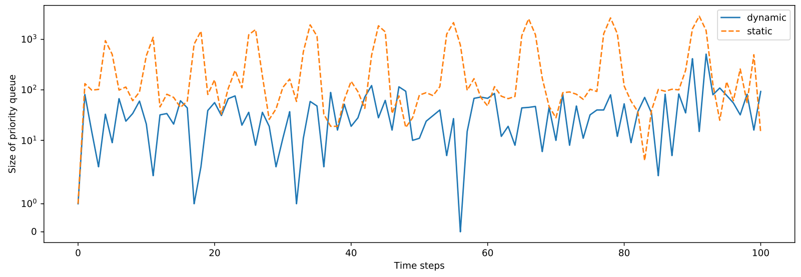


Fig. 6: Size of priority queue for static and dynamic algorithms for the SLASHDOT graph for a batch size of 100.

Finally, Figure 6 plots the sizes of the priority queues in both algorithms for the SLASHDOT graph. We sample at 100 evenly spaced time points throughout the duration of both algorithms and plot the size of the priority queue at that time point for the dynamic (solid orange line) and static (dotted blue line) algorithms. The size of the priority queue in the dynamic algorithm consistently remains orders of magnitude lower than the size of the priority queue in the static algorithm. The periodic spikes in the size of the priority queue in the static algorithm can be attributed to the periods after processing an element where we add all the neighbors of the currently processed vertex to the priority queue.

5 Conclusions

This paper presented a new algorithm for computing the values of personalized nonbacktracking walk-based centrality scores of the vertices in both static and dynamic graphs. The algorithm returns approximations of scores by counting NBTWs up to a certain length starting at a given seed vertex. In past literature, these centrality values have been computed using a linear algebraic formulation and only on static graphs. Our algorithm agglomeratively counts NBTWs in graphs to obtain the corresponding centrality scores and for static graphs the results presented indicate that our method obtains good quality approximations of the scores compared to a linear algebraic computation. For dynamic graphs, our algorithm is able to avoid a full static recomputation and efficiently computes updated scores, given edge updates to the graph. Our dynamic algorithm returns exactly the same scores as the static algorithm, meaning we have no approximation error. Furthermore, our dynamic algorithm is several orders of magnitude faster than the static algorithm, indicating our approach has large performance benefits.

References

1. Peter Grindrod, Desmond J Higham, and Vanni Noferini. The deformed graph laplacian and its applications to network centrality analysis. *SIAM Journal on Matrix Analysis and Applications*, 39(1):310–341, 2018.
2. Travis Martin, Xiao Zhang, and MEJ Newman. Localization and centrality in networks. *Physical review E*, 90(5):052808, 2014.
3. Tatsuro Kawamoto. Localized eigenvectors of the non-backtracking matrix. *Journal of Statistical Mechanics: Theory and Experiment*, 2016(2):023404, 2016.
4. Dragos Cvetkovic, Dragoš M Cvetković, Peter Rowlinson, and Slobodan Simic. *Eigenspaces of graphs*, volume 66. Cambridge University Press, 1997.
5. Michele Benzi, Ernesto Estrada, and Christine Klymko. Ranking hubs and authorities using matrix functions. *Linear Algebra and its Applications*, 438(5):2447–2474, 2013.
6. Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.
7. Phillip Bonacich. Some unique properties of eigenvector centrality. *Social networks*, 29(4):555–564, 2007.
8. David F Gleich. Pagerank beyond the web. *SIAM Review*, 57(3):321–363, 2015.
9. Mary Aprahamian, Desmond J Higham, and Nicholas J Higham. Matching exponential-based and resolvent-based centrality measures. *Journal of Complex Networks*, page cnv016, 2015.
10. Ernesto Estrada and Juan A Rodriguez-Velazquez. Subgraph centrality in complex networks. *Physical Review E*, 71(5):056103, 2005.
11. Michele Benzi and Christine Klymko. Total communicability as a centrality measure. *Journal of Complex Networks*, 1(2):124–149, 2013.
12. Francesca Arrigo, Peter Grindrod, Desmond J Higham, and Vanni Noferini. Non-backtracking walk centrality for directed networks. *Journal of Complex Networks*, 6(1):54–78, 2017.
13. Ulrik Brandes and Christian Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007.
14. Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.
15. Florent Krzakala, Cristopher Moore, Elchanan Mossel, Joe Neeman, Allan Sly, Lenka Zdeborová, and Pan Zhang. Spectral redemption in clustering sparse networks. *Proceedings of the National Academy of Sciences*, 110(52):20935–20940, 2013.
16. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
17. Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.
18. Réka Albert, Hawoong Jeong, and Albert-László Barabási. Internet: Diameter of the world-wide web. *Nature*, 401(6749):130–131, 1999.