Social Network Analysis and Mining (2018) 8:26
https://doi.org/10.1007/s13278-018-0504-3

ORIGINAL ARTICLE

# Incrementally updating Katz centrality in dynamic graphs

Eisha Nathan[1] · David A. Bader[1]

Received: 1 November 2017 / Revised: 21 March 2018 / Accepted: 22 March 2018 / Published online: 29 March 2018

**Abstract**
A variety of large datasets, such as social networks or biological data, can be represented as graphs. A common query in graph analysis is to identify the most important vertices in a graph. Centrality metrics are used to obtain numerical scores for each vertex in the graph. The scores are then translated to rankings identifying relative importance of vertices. In this work, we focus on Katz centrality, a linear algebra-based metric. In many real applications, since data are constantly being produced and changed, it is necessary to have a dynamic algorithm to update centrality scores with minimal computation when the graph changes. We present an algorithm for updating Katz centrality scores in a dynamic graph that incrementally updates the centrality scores as the underlying graph changes. Our proposed method exploits properties of iterative solvers to obtain updated Katz scores in dynamic graphs. Our dynamic algorithm improves performance and achieves speedups of over two orders of magnitude compared to a standard static algorithm while maintaining high quality of results.

**Keywords** Katz centrality · Dynamic graphs · Iterative solvers

## 1 Introduction

Graphs are a natural representation for modeling relationships between entities, in web traffic, financial transactions, computer networks, or society (Benzi and Klymko 2014). A significant question arising from the analysis of graphs is to identify the most important vertices in a graph (Kempe et al. 2003). Vertex importance is termed as *centrality* and centrality scores can be used to provide *rankings* on the vertices of a graph (Benzi et al. 2013). Consider a web-Google graph. When inputting a search query into Google, a user typically wants the most relevant results to the search query to show up at the top of the returned results. Furthermore, a user likely only has enough human resources to examine the top 75–100 results. Therefore, correct ranking is important with respect to the search results. In a network modeling disease spread, an analyst might wish to find the sites of disease origin. These queries are answered by looking at the highly ranked vertices.

✉ Eisha Nathan
enathan3@gatech.edu

David A. Bader
bader@gatech.edu

1 School of Computational Science and Engineering, Georgia Tech, Atlanta, GA, USA

In real-world networks today, new data are constantly being produced, leading to the notion of dynamic graphs. Dynamic graph data can represent the changing relationships in networks. For example, consider a graph modeling relationships on Facebook, where vertices are people and edges exist between two vertices if the corresponding people are friends on Facebook. As new friendships are formed and old ones deleted, the corresponding graph will change over time to reflect these new relationships. The identification of central vertices in an evolving network is a fundamental problem in network analysis (Benzi et al. (2013)). Development of dynamic algorithms for updating centrality measures in graphs is therefore an important research problem. A naive method of obtaining updated centrality scores in dynamic graphs is to recalculate the scores from scratch every time the graph is changed. We refer to this simplistic method as *static recomputation*. However, this becomes computationally intensive to constantly recalculate from scratch as more and more data are added to the graph. Therefore, it is preferable to have alternate methods to efficiently obtain updated centrality scores in a changing graph. In this work, we present a new algorithm for updating Katz centrality in dynamic graphs. Katz centrality is a metric that measures the affinity between vertices as a weighted sum of the walks between them while penalizing longer walks in the network (Katz 1953). The linear algebraic formulation of Katz centrality lends itself to a dynamic algorithm based in a numerical linear algebra environment using iterative solvers. In contrast to a static algorithm that is run once on an unchanging graph, our algorithm is able to incrementally update solutions between different timepoints as new data are added to a changing graph.

### 1.1 Contributions

We present a new linear algebra-based method to incrementally update Katz centrality scores in a dynamic graph. Our algorithm is faster than recomputing centrality scores from scratch every time the graph is updated and returns high-quality results that are similar to results obtained with a simple static recomputation method. This paper presents the extended version of the work by Nathan and Bader (2017). We evaluate our algorithm on larger datasets and present an alternate approach and discuss its shortcomings compared to our algorithm. We examine how our algorithm behaves with respect to both global and personalized centrality scores and analyze how the granularity of the time step affects the quality of our algorithm. We compare our dynamic algorithm to multiple static recomputation methods and additionally examine the quality of our algorithm if we are only concerned with recall of the highly ranked vertices in dynamic graphs. Finally, we present an approach on how to handle vertex additions and deletions using our algorithm.

Section 2 discusses relevant work in the literature, and Sect. 3 provides the necessary background and definitions required to understand our work. In Sect. 4, we present an alternate method and provide the motivation for our dynamic algorithm. We present our algorithm for updating Katz centrality scores in dynamic graphs in Sect. 5. Section 6 provides an analysis of our method on both synthetic and real-world networks with respect to performance and quality. In Sect. 7, we discuss a possible approach for handling vertex additions and deletions and in Sect. 8 we conclude.

## 2 Related work

Betweenness and closeness centrality are two very popular graph metrics in network analysis for identifying the most important vertices in a graph, with specific applications in network stability, traffic predictions, and social network analysis (Benzi et al. 2013). Betweenness centrality ($BC(v)$) looks at the vertices with high betweenness, i.e., those vertices whose removal would cause a significant number of shortest paths do not exist anymore. This notion was first established by Freeman, to compare the number of shortest paths going through a vertex $v$ with the total number of shortest paths (Freeman 1977). Formally it is defined as $BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where $\sigma_{st}(v)$ is the number of shortest paths from vertex $s$ to vertex $t$ that include node $v$ and $\sigma_{st}$ is the number of shortest paths from $s$ to $t$ in general. Closeness centrality ($CC(v)$) was first introduced by Bavelas in 1950 to measure the 'farness' of a vertex, defined as the sum of its distances from all other vertices, and its 'closeness,' defined as the reciprocal of the farness (Bavelas 1950). Closeness centrality measures how close a vertex is to all other vertices based on the shortest-path length. It is formally defined as $CC(v) = \frac{1}{\sum_{t \in V} d_G(v,t)}$, where $d_G(v, t)$ is the length of the shortest path between node $v$ and node $t$. Since both these metrics are fairly computationally intensive to calculate, in the case of dynamic graphs it is optimal to have an algorithm that can update the centrality values with minimal effort as the graph updates instead of recomputing the centrality values from scratch. In Wei and Carley (2014), the authors propose an algorithm to update both betweenness and closeness calculations together after receiving edge updates to the graph. By splitting up the calculation of the centrality metrics into two parts, they avoid performing unnecessary calculations performed in previous time steps. The first step repeats a calculation process until the shortest path is converged, and the second step aggregates the shortest path calculation into closeness and betweenness centralities. The first step can be performed for both closeness and betweenness centrality simultaneously. The authors in Sariyuce et al. (2013) propose an incremental algorithm for closeness centrality by exploiting specific network topological properties: specifically their shortest-distance distributions, biconnected components distributions, and the existence of vertices with identical neighborhoods. They achieve a mean speedup of 43.5× for smaller graphs with less than 500 K edges and 99.8× for larger graphs with more than 500 K edges. Finally, the authors in Green et al. (2012) propose an incremental algorithm for updating betweenness centrality values by maintaining additional data structures to store previously computed values. They are able to achieve speedups of 100–400× on synthetic networks and speedups of 36–148× on real networks.

Several centrality measures can be expressed as functions of the adjacency matrix of a graph (Benzi et al. 2013). The centrality metric is obtained by solving a linear system, and the solution is then a vector consisting of a number for each vertex in the graph identifying its relative importance. Obtaining an exact solution via direct methods is prohibitively computationally expensive, since we are typically required to take the inverse of a matrix. The most accurate way to obtain the exact solution would be by LU decomposition, which costs $\mathcal{O}(n^2)$, where $n$ is the number of vertices in the graph. In many real networks, the amount of data is massive and $n$ can be as large as millions or billions of vertices, so direct methods such as these do not scale and are impractical. Moreover, there is no technique to compute an exact solution for a general graph in finite precision arithmetic, so in practice, iterative methods are often used to obtain an approximate solution. We explain this in more detail in Sect. 3.1.

PageRank is a common method for ranking vertices in graphs, where a high score means random walks through the graph tend to visit the highly ranked vertices, and was first introduced rank webpages in a web search (Page et al. 1999). Given a search term from the user, PageRank incorporates a measure of a webpage's importance into the results of a set of webpages that could be relevant to the desired search term. However, over time many more applications have risen, such as in bibliometrics, social, and information network analysis. For example, personalized PageRank vectors have been used for local community detection (Riedy 2016). It has also been used in analysis of road networks and for link prediction and recommendation systems (Gleich 2015). To define the PageRank problem, we consider a random surfer model: a hypothetical random web surfer navigating between webpages online. When this random surfer visits a webpage, he tosses a coin; if the coin comes up heads he randomly clicks on a link from the current page and transitions there, if the coin comes up tails, he *teleports* to a (possibly random) page independent of the current page's identity. Let $P = A^T D^{-1}$ be the transition matrix of probabilities, specifically $P(i, j)$ is the probability of transitioning from page $j$ to page $i$. Assume the random surfer transitions according to the link structure of the web with probability $\alpha$ and teleports randomly with probability $1 - \alpha$. When teleporting randomly, the surfer teleports according to a teleportation distribution vector $\mathbf{v}$, where $\mathbf{v}$ is typically a uniform distribution over all pages. Many applications typically set $\alpha$ to 0.85. Then, the solution $\mathbf{x}$ to the equation $(I - \alpha P)\mathbf{x} = (1 - \alpha)\mathbf{v}$ gives the desired PageRank vector.

There has been much work in the literature for updating PageRank for dynamic graphs, and these techniques fall under two general areas: (1) linear algebraic methods that mainly use techniques from linear and matrix algebra (Chen et al. 2004; Chien et al. 2001) and (2) Monte Carlo methods that use a small number of simulated random walks per vertex (Sarma et al. 2011; Gyöngyi et al. 2004). "Aggregation" methods operate under the assumption that changes to the graph will affect only a localized portion of the PageRank vector (Langville and Meyer 2002, 2004). These methods partition the set of vertices into two disjoint sets: $C$ is the set of all vertices close to/affected by the changes made to the underlying network and $V \backslash C$ consists of the remaining vertices. The vertices in $V \backslash C$ are aggregated into a single hyper-vertex and a smaller graph is created. Using the smaller graph, the PageRank values of all vertices are updated and the result is pushed back to the initial larger graphs. While initially accurate for the first few edge updates, since this method ultimately produces an approximation to the exact PageRank vector, there is the possibility that the error could accumulate over time. More importantly, these types of techniques do not fare very well for real-time monitoring applications in terms of performance and can end up being very slow. The second class of techniques rely on Monte Carlo methods for the incremental computation of random walk methods such as PageRank (Bahmani et al. 2010). While they are far more efficient for performance and produce better quality results, thus far these techniques have only been studies in static networks. These methods maintain a small number of short random walk segments starting at each vertex in the graph. For the case of identifying the top $k$ vertices, these methods are able to provide highly accurate estimates of the centrality values for the top vertices, but smaller values in the personalized case are nearly identical and therefore impossible to tell apart. Finally, Riedy (2016) provides a formula for updating PageRank using the notion of iterative refinement to update the residual of the linear system after receiving graph updates. Katz centrality is a similar linear algebraic centrality metric to PageRank; however, while there is much work on dynamic PageRank in the literature, as far as the authors are aware there is no work on dynamic Katz centrality. Therefore, in this work, we focus on Katz centrality and develop an algorithm for updating scores on vertices in a dynamic graph.

## 3 Background and definitions

Many data analysis problems are phrased as numerical problems for a more tractable solution (Kokiopoulou et al. 2011). In this work, we use a linear algebra-based method to compute Katz centrality to obtain updated centrality scores on the vertices of a dynamic graph.

Let $G = (V, E)$ be a graph, where $V$ is the set of $n$ vertices and $E$ the set of $m$ edges. Denote the $n \times n$ adjacency matrix $A$ of $G$ as

$$A(i,j) = \begin{cases} 1, & \text{if } (i,j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

We use undirected, unweighted graphs so $\forall i, j$, $A(i,j) = A(j, i)$ and all edge weights are 1. A dynamic graph can change over time due to edge insertions and deletions and vertex additions and deletions. As a graph changes, we can take snapshots of its current state. We denote the current snapshot of the dynamic graph $G$ and corresponding adjacency matrix $A$ at time $t$ by $G_t = (V_t, E_t)$ and $A_t$, respectively. In this work, the vertex set is constant over time so $\forall t, V_t = V$, and we deal only with edge insertions, although our algorithm can be applied for edge deletions as well. Given edge updates to the graph, we write the new adjacency matrix at time $t + 1$ as $A_{t+1} = A_t + \Delta A$, where $\Delta A$ represents the new edges being added into the graph.

Katz centrality scores count the number of weighted walks in a graph between vertices in a graph, while penalizing longer walks in the network by a user-chosen parameter $\alpha$. A walk of length $k$ in a graph is a sequence of $k$ vertices $v_1, v_2, \dots, v_k$ where both vertices and edges are allowed to repeat. Counts of walks in a graph can be calculated using powers of the adjacency matrix (Higham 2008). Specifically, $A^k(i,j)$ represents the number of walks of length $k$ from vertex $i$ to $j$. To obtain weighted counts of walks of all lengths in the network, we can derive the following infinite series:

$$\sum_{i=0}^{\infty} \alpha^{i-1}A^i = A + \alpha A^2 + \alpha^2 A^3 + \cdots + \alpha^{k-1}A^k + \cdots .$$

This infinite series converges to the matrix resolvent $A(I - \alpha A)^{-1}$. Katz originally used the row sums of this matrix to calculate centrality scores as $A(I - \alpha A)^{-1}\mathbf{1}$. The result is an $n \times 1$ vector where the $j$th value in this vector represents the total number of weighted walks of all lengths starting at vertex $j$. We refer to these as *global Katz scores*. Similarly, we can derive a corresponding formula for *personalized Katz scores* as $A(I - \alpha A)^{-1}\mathbf{e}_i$, where $\mathbf{e}_i$ is the $i$th canonical basis vector. The result is again an $n \times 1$ vector, where the $j$th value in this vector represents the number of weighted walks of all lengths starting at vertex $i$ and ending at vertex $j$. We set $\alpha = 0.85/\|A\|_2$ to mimic PageRank computations (Gleich 2015), and in this work we study both global and personalized scores.

### 3.1 Iterative methods

Directly solving for the exact Katz centrality scores $\mathbf{c}$ is on the order of $\mathcal{O}(n^2)$ and quickly becomes very expensive and impractical as $n$ grows large. Therefore, in practice we use iterative methods to obtain an approximation which costs $\mathcal{O}(m)$ provided the number of iterations is not very large. Many real-world graphs are sparse and $m \ll n^2$ (Albert et al. 1999). Iterative methods approximate the solution $\mathbf{x}$ to a linear system $M\mathbf{x} = \mathbf{b}$, given $M$ and $\mathbf{b}$ by starting with an initial guess $\mathbf{x}^{(0)}$ and improving the current guess with each iteration until some stopping criterion is reached. This stopping criterion can be a predetermined number of iterations, a desired level of accuracy, or some application-specific terminating criterion. At each iteration $k$ of the iterative solver, we obtain a new approximation $\mathbf{x}^{(k)}$. Unless otherwise stated, all the work here assumes a starting approximation $\mathbf{x}^{(0)}$ as the all zeros vector, although any starting vector can be chosen. The residual at the $k$th iteration is denoted as $\mathbf{r}^{(k)} = \mathbf{b} - M\mathbf{x}^{(k)}$ and is a measure of how close the current solution $\mathbf{x}^{(k)}$ is to solve the system $M\mathbf{x} = \mathbf{b}$. We let $M = I - \alpha A$, so we solve the linear system $M\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$ using an iterative method and then obtain the Katz scores using a matrix-vector multiplication in $\mathcal{O}(m)$ as $\mathbf{c} = A\mathbf{x}$. We set $\mathbf{b} = \mathbf{1}$ for the global scores and $\mathbf{b} = \mathbf{e}_i$ for the personalized scores. The iterative method we use here is the Jacobi algorithm (Saad 2003) outlined in Algorithm 1. Here, $D$ is the matrix consisting of the diagonal entries from $M$ and $R$ is the matrix of all off-diagonal entries of $M$. We terminate the solver when the solution changes by less than a fixed tolerance *tol* (Riedy 2016), or when $\|\mathbf{x}^{k+1} - \mathbf{x}^k\|_2$.

**Algorithm 1** Solve $M\mathbf{x} = \mathbf{b}$ to tolerance *tol* using Jacobi algorithm.

```
1: procedure JACOBI($M, \mathbf{b}, tol$)
2:     $k = 0$
3:     $\mathbf{x}^{(0)} = \mathbf{0}$
4:     $D = diag(M)$
5:     $R = M - D$
6:     while $\|\mathbf{x}^{k+1} - \mathbf{x}^k\|_2 > tol$ do
7:         $\mathbf{x}^{(k+1)} = D^{-1}(R\mathbf{x}^{(k)} + \mathbf{b})$
8:         $k += 1$
       return $\mathbf{x}^{(k+1)}$
```

Our dynamic algorithm is also motivated by principles of iterative refinement, another iterative method that adds a correction to the current guess to obtain a more accurate approximation (Wilkinson 1994). To compute the solution $\mathbf{x}$ to the linear system $M\mathbf{x} = \mathbf{b}$, iterative refinement repeatedly performs the following steps at each iteration $k$.

1. Compute residual $\mathbf{r}^{(k)} = \mathbf{b} - M\mathbf{x}^{(k)}$
2. Solve system $M\mathbf{d}^{(k)} = \mathbf{r}^{(k)}$ for correction $\mathbf{d}^{(k)}$
3. Add correction to obtain new solution $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{d}^{(k)}$

Note that we can use any other iterative method to solve the system in Step 2.

## 4 Motivation and initial approach

### 4.1 Static algorithm

Given edge updates to the graph, the static algorithm to recompute the Katz centrality scores in the updated graph first calculates $\mathbf{x}$ from scratch using an iterative method and then calculates $\mathbf{c}$ using a single matrix-vector multiplication. This procedure is given in Algorithm 2 to obtain the new solution $\mathbf{c}_{t+1}$ at time $t + 1$ given updates $\Delta A$ to the graph. After a batch of edges has been inserted into the network, the adjacency matrix is updated to $A_{t+1}$ and the vector $\mathbf{x}_{t+1}$ is recomputed using the Jacobi method from Algorithm 1.

**Algorithm 2** Solve for $\mathbf{c}_{t+1}$ at time $t+1$ given new edge updates $\Delta A$.

```
1: procedure STATIC_KATZ($A_t, \Delta A$)
2:     $A_{t+1} = A_t + \Delta A$                          ▷ Updated adjacency matrix
3:     $M_{t+1} = I - \alpha A_{t+1}$                     ▷ New linear system
4:     $\mathbf{x}_{t+1} = $ JACOBI($M_{t+1}, \mathbf{1}, 10^{-4}$)   ▷ Recomputed vector
5:     $\mathbf{c}_{t+1} = A_{t+1}\mathbf{x}_{t+1}$       ▷ New Katz scores
6: return $\mathbf{c}_{t+1}$
```

Since calculating $\mathbf{c}_t$ given $\mathbf{x}_t$ at any timepoint $t$ is one matrix-vector multiplication and can be done in $\mathcal{O}(m)$, this is not the bottleneck of the static algorithm. As more data are added to the graph, the number of iterations taken to update $\mathbf{x}_{t+1}$ in Line 4 increases and pure recomputation becomes increasingly expensive as the graph size increases. We thus focus the development of our dynamic algorithm on limiting the number of iterations taken to obtain the updated vector $\mathbf{x}_{t+1}$. Calculating $\mathbf{c}$ is the same in the static and our dynamic algorithm and so for the rest of the paper we focus our discussions on the vector $\mathbf{x}$.

## 4.2 Motivation

In many low-latency applications, the number of edge updates, or equivalently, the size of $\Delta A$, is significantly smaller than the size of the entire graph $A$. If the change $\Delta A$ is small relative to the size of the graph, the new graph will be similar to the old graph. It follows that the new solution $\mathbf{x}_{t+1}$ at time $t+1$ might be similar to the old solution $\mathbf{x}_t$ at time $t$. This is the intuition behind our dynamic algorithm. Figure 1 plots the difference between subsequent solutions for global scores each time the graph changes for the Facebook graph. The $x$ axis simulates time as more edges are being added into the graph. We insert 1000 edges into the graph at each time step. The $y$ axis is the 2-norm difference between solutions at consecutive timepoints, $\|\mathbf{x}_{t+1} - \mathbf{x}_t\|_2$. Since the Katz scores themselves can be as high as $10^4$, a difference of $10^{-1}$ across insertions of edges over time is relatively small. This indicates that the solutions across timepoints are not very different, suggesting that the static algorithm of recomputing the centrality metric from scratch is doing a lot of unnecessary work. Therefore, our dynamic algorithm therefore only targets places in the vector that are affected by updates to the graph.

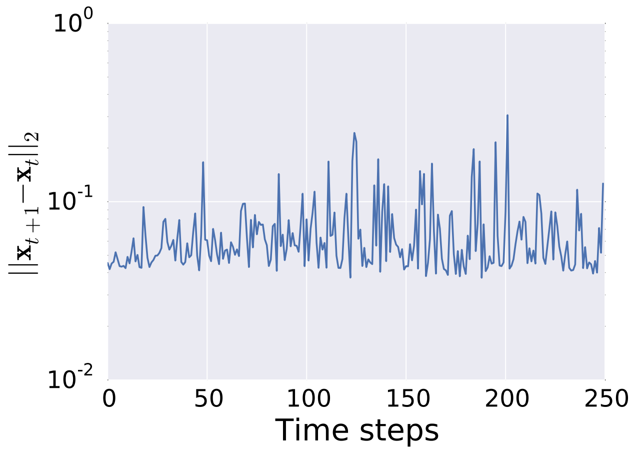


**Fig. 1** Difference in consecutive solutions over time. Small changes in solutions suggest a dynamic algorithm could work by applying incremental updates to previous solutions

## 4.3 Initial approach

Here we present a "first-pass" algorithm and discuss its shortcomings. This provides the motivation for the development of our dynamic algorithm in Sect. 5. Suppose we have the solution $\mathbf{x}_t$ for the adjacency matrix $A_t$ at a specific timepoint $t$. We want to solve for the new solution at time $t+1$ as $\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{\Delta x}$. Given edge updates to the graph, we want to solve for the vector $\mathbf{x}_{t+1}$ in the linear system $(I - \alpha A_{t+1})\mathbf{x}_{t+1} = \mathbf{1}$ for the global scores, or $(I - \alpha A_{t+1})\mathbf{x}_{t+1} = \mathbf{e}_i$ for the personalized scores equivalently. Using basic algebra, we can rearrange the terms in the linear system to derive an iterative update as follows:

$$
\begin{aligned}
\mathbf{1} &= (I - \alpha A_{t+1})\mathbf{x}_{t+1} \\
\mathbf{1} &= (I - \alpha A_{t+1})(\mathbf{x}_t + \mathbf{\Delta x}) \\
\mathbf{1} &= (I - \alpha A_{t+1})\mathbf{x}_t + (I - \alpha A_{t+1})\mathbf{\Delta x} \\
\mathbf{1} &= \mathbf{x}_t - \alpha(A_t + \Delta A)\mathbf{x}_t + \mathbf{\Delta x} - \alpha A_{t+1}\mathbf{\Delta x} \\
\mathbf{1} &= \mathbf{x}_t - \alpha A_t\mathbf{x}_t - \alpha \Delta A\mathbf{x}_t + \mathbf{\Delta x} - \alpha A_{t+1}\mathbf{\Delta x} \\
\mathbf{1} &= (I - \alpha A_t)\mathbf{x}_t - \alpha \Delta A\mathbf{x}_t + \mathbf{\Delta x} - \alpha A_{t+1}\mathbf{\Delta x}
\end{aligned}
$$

Since $(I - \alpha A_t)\mathbf{x}_t = \mathbf{1}$, we can split and rearrange the terms as

$$\mathbf{\Delta x} = \alpha A_{t+1}\mathbf{\Delta x} + \alpha \Delta A\mathbf{x}_t, \tag{1}$$

and turn this into an iterative update to solve for $\mathbf{\Delta x}$:

$$\mathbf{\Delta x}^{(k+1)} = \alpha A_{t+1}\mathbf{\Delta x}^{(k)} + \alpha \Delta A\mathbf{x}_t \tag{2}$$

However, this simplistic approach tends to accumulate error instead of converging to the same solution as static recomputation. We provide a more in-depth analysis of the quality of this alternate method in Sect. 6. This approach (henceforth referred to as the "alternate" method) is based off of a forward error analysis. Forward error of an algorithm tells us the difference between the result we obtain and the actual solution (Higham 2002). In our case for an exact solution $\mathbf{x}^*$, the forward error is the quantity $\mathbf{x}^* - \mathbf{x}_{t+1}$ where $\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{\Delta x}$ as derived above in Eq. 2. By aiming to minimize this quantity, we will see that the error tends to unfortunately rapidly increase. Therefore, we next present our dynamic algorithm based off of a backward error analysis in Sect. 5. Backward error is the smallest quantity $\delta$ such that $(I - \alpha A_{t+1})(\mathbf{x}_{t+1} + \delta) = \mathbf{1}$, or in other words, we obtain an understanding on what problem we actually solved.

## 5 Dynamic algorithm

Our dynamic algorithm computes the correction $\mathbf{\Delta x}$, the difference in the solutions at timepoints $t$ and $t+1$, using principles of iterative refinement. For the purposes of deriving the algorithm, we do so w.r.t. the global scores. For personalized scores w.r.t. vertex $i$, we simply replace the vector $\mathbf{1}$ with $\mathbf{e}_i$. Since we use the old solution as a starting point for the new solution, we first measure how close the old solution is to solve the system for the new graph. We do so by introducing the concept of an "approximate residual" denoted as $\widetilde{\mathbf{r}}_{t+1}$. This can be written in terms of the current residual at time $t$, $\mathbf{r}_t = \mathbf{1} - M_t\mathbf{x}_t$, edge updates $\Delta A$, and the old solution $\mathbf{x}_t$. The algorithm to compute $\widetilde{\mathbf{r}}_{t+1}$ is given in Algorithm 3 with the corresponding proof of correctness in Theorem 1.

**Algorithm 3** Solve for approximate residual $\widetilde{\mathbf{r}}_{t+1}$ at time $t+1$.

```
1: procedure GET_APPROXIMATE_RESIDUAL($\Delta A, \mathbf{r}_t, \mathbf{x}_t$)
2:     $\widetilde{\mathbf{r}}_{t+1} = \mathbf{r}_t + \alpha \Delta A \mathbf{x}_t$
3: return $\widetilde{\mathbf{r}}_{t+1}$
```

**Theorem 1** *Algorithm 3 correctly calculates the approximate residual at time* $t+1$.

**Proof** The approximate residual $\widetilde{\mathbf{r}}_{t+1}$ measures how close the current solution $\mathbf{x}_t$ is to solve the updated system $A_{t+1}$.

$$\begin{aligned}\widetilde{\mathbf{r}}_{t+1} &= \mathbf{1} - M_{t+1}\mathbf{x}_t\\ &= \mathbf{1} - (I - \alpha A_{t+1})\mathbf{x}_t\\ &= \mathbf{1} - \mathbf{x}_t + \alpha A_{t+1}\mathbf{x}_t\\ &= \mathbf{1} - \mathbf{x}_t + \alpha A_t\mathbf{x}_t - \alpha A_t\mathbf{x}_t + \alpha A_{t+1}\mathbf{x}_t\\ &= \mathbf{r}_t + \alpha(A_{t+1} - A_t)\mathbf{x}_t\\ &= \mathbf{r}_t + \alpha\Delta A\mathbf{x}_t\end{aligned}$$

□

We then use the approximate residual $\widetilde{\mathbf{r}}_{t+1}$ to solve a linear system for the correction $\boldsymbol{\Delta x}$. Solved exactly, this linear system will give the same scores as static recomputation, but solved to some preset tolerance as discussed earlier, it will provide a good quality approximation of the updated centrality scores. We examine the effect of varying the tolerance on the performance of our dynamic algorithm in Sect. 6. This procedure and the corresponding proof of correctness are given in Algorithm 4 and Theorem 2, respectively.

**Algorithm 4** Use iterative refinement to obtain $\boldsymbol{\Delta}\mathbf{x}$.

```
1: procedure OBTAIN_DEL_X($A_{t+1}, \widetilde{\mathbf{r}}_{t+1}$)
2:     $\boldsymbol{\Delta}\mathbf{x}$ = JACOBI($I - \alpha A_{t+1}, \widetilde{\mathbf{r}}_{t+1}, 10^{-4}$)
3: return $\boldsymbol{\Delta}\mathbf{x}$
```

**Theorem 2** *Algorithm 4 correctly calculates the correction* $\boldsymbol{\Delta x}$ at time $t+1$.

**Proof** Since the approximation residual $\widetilde{\mathbf{r}}_{t+1}$ measures how close the current solution is to the solution of the updated system, we use $\widetilde{\mathbf{r}}_{t+1}$ to solve for the correction $\boldsymbol{\Delta x}$ using principles of iterative refinement.

$$\begin{aligned}(I - \alpha A_{t+1})\boldsymbol{\Delta x} &= \widetilde{\mathbf{r}}_{t+1} = \mathbf{r}_t + \alpha\Delta A\mathbf{x}_t\\ \boldsymbol{\Delta x} - \alpha A_{t+1}\boldsymbol{\Delta x} &= \mathbf{r}_t + \alpha\Delta A\mathbf{x}_t\end{aligned}$$

We can turn this into an iterative update:

$$\boldsymbol{\Delta x}^{(k+1)} = \alpha A_{t+1}\boldsymbol{\Delta x}^{(k)} + \alpha\Delta A\mathbf{x}_t + \mathbf{r}_t$$

This formulation lends itself quite nicely to using the Jacobi algorithm. □

The final step of our algorithm is to update the residual $\mathbf{r}_t$ for the next timepoint. We do so by calculating $\Delta\mathbf{r}$, the difference in the two residuals at time $t$ and $t+1$. This procedure is given in Algorithm 5 with the corresponding proof of correctness in Theorem 3.

**Algorithm 5** Updating residual at time $t+1$.

```
1: procedure UPDATE_RESIDUAL($A_{t+1}, \Delta A, \mathbf{x}_{t+1}$)
2:     $\Delta\mathbf{r} = \alpha\Delta A\mathbf{x}_t - (I - \alpha A_{t+1})\boldsymbol{\Delta}\mathbf{x}$
3: return $\Delta\mathbf{r}$
```

**Theorem 3** Algorithm 5 correctly updates the residual at time $t+1$.

**Proof** The residual $\mathbf{r}_{t+1}$ at time $t+1$ measures the correctness of the updated solution $\mathbf{x}_{t+1}$. We write the new residual $\mathbf{r}_{t+1}$ in terms of the old residual $\mathbf{r}_t$ to obtain the difference between the two as $\Delta\mathbf{r}$.

$$\begin{aligned}\mathbf{r}_{t+1} &= \mathbf{1} - (I - \alpha A_{t+1})\mathbf{x}_{t+1}\\ &= \mathbf{1} - (I - \alpha A_{t+1})(\mathbf{x}_t + \boldsymbol{\Delta x})\\ &= \mathbf{1} - (I - \alpha A_{t+1})\mathbf{x}_t - (I - \alpha A_{t+1})\boldsymbol{\Delta x}\\ &= \widetilde{\mathbf{r}}_{t+1} - (I - \alpha A_{t+1})\boldsymbol{\Delta x}\\ &= \mathbf{r}_t + \alpha\Delta A\mathbf{x}_t - (I - \alpha A_{t+1})\boldsymbol{\Delta x}\\ &= \mathbf{r}_t + \Delta\mathbf{r}\\ \therefore \boldsymbol{\Delta}\mathbf{r} &= \alpha\Delta A\mathbf{x}_t - (I - \alpha A_{t+1})\boldsymbol{\Delta x}\end{aligned}$$

□

The entire procedure for updating Katz centrality scores in a dynamic graph is outlined in Algorithm 6, DYNAMIC_KATZ, and uses the three previously described subroutines. First, in line 2 we calculate the current residual $\mathbf{r}_t$, which is easily obtained given the current snapshot of the graph $A_t$ and solution $\mathbf{x}_t$ at time $t$. In line 3, we form the new snapshot of the graph $A_{t+1}$ using the new batches of edges that are being inserted into the graph. In line 4, we call the first subroutine GET_APPROXIMATE_RESIDUAL, Algorithm 3, to return the approximate residual $\widetilde{\mathbf{r}}_{t+1}$. Next in line 5 we solve for the difference $\boldsymbol{\Delta x}$ between the vectors $\mathbf{x}_{t+1}$ and $\mathbf{x}_t$ using the subroutine OBTAIN_DEL_X, Algorithm 4. In line 6, we calculate the new solution $\mathbf{x}_{t+1}$ using the old solution $\mathbf{x}_t$ and the calculated correction $\boldsymbol{\Delta x}$. Finally, after updating the solution from time $t$ to the solution at $t+1$, lines 7 and 8 update the residual between these two timepoints using the subroutine UPDATE_RESIDUAL in Algorithm 5. Finally, at the end of the procedure in line 9 we return the new solution $\mathbf{x}_{t+1}$.

**Algorithm 6** Solve for $\mathbf{x}_{t+1}$ at time $t+1$ given previous solution $\mathbf{x}_t$ at time $t$ and new edge updates $\Delta A$.

```
1: procedure DYNAMIC_KATZ($A_t, \mathbf{x}_t, \Delta A$)
2:     $\mathbf{r}_t = \mathbf{1} - (I - \alpha A_t)\mathbf{x}_t = \mathbf{1} - \mathbf{x}_t + \alpha A_t$
3:     $A_{t+1} = A_t + \Delta A$
4:     $\tilde{\mathbf{r}}_{t+1}$ = GET_APPROXIMATE_RESIDUAL($\Delta A, \mathbf{r}_t, \mathbf{x}_t$)
5:     $\boldsymbol{\Delta}\mathbf{x}$ = OBTAIN_DEL_X($A_{t+1}, \tilde{\mathbf{r}}_{t+1}$)
6:     $\mathbf{x}_{t+1} = \mathbf{x}_t + \boldsymbol{\Delta}\mathbf{x}$
7:     $\boldsymbol{\Delta}\mathbf{r}$ = UPDATE_RESIDUAL($A_{t+1}, \Delta A, \mathbf{x}_{t+1}$)
8:     $\mathbf{r}_{t+1} = \mathbf{r}_t + \boldsymbol{\Delta}\mathbf{r}$
9: return $\mathbf{x}_{t+1}$
```

Note that while in this paper we only examine edge insertions in a dynamic graph, the algorithm is equally well suited for handling edge deletions. Changes to the graph are represented by $\Delta A$. If we are inserting edge $(i, j)$ into the graph at time $t$, we set $\Delta A(i,j) = 1$. Similarly if we want to delete edge $(i, j)$, we can write $\Delta A(i,j) = -1$.

### 5.1 Complexity analysis

The majority of the work done by the dynamic algorithm is in Algorithm 4 (OBTAIN_DEL_X). Since we still require a matrix-vector multiplication by $A_{t+1}$ at the end of the algorithm, the worst-case complexity of the dynamic algorithm is the same as static recomputation and is $\mathcal{O}(m)$, apart from a constant (based on the number of iterations taken by the iterative solver). However, in practice we observe that we are able to obtain significant speedups in both time and iterations compared to static recomputation while maintaining a good quality of results returned. This is due to the fact that the number of iterations taken by our dynamic algorithm is far fewer than that of static recomputation and we are able to converge to the solution faster.

## 6 Results

### 6.1 Experimental setup

We test our method of updating Katz centrality scores in dynamic graphs on both synthetic and real-world networks. For synthetic networks, we use Erdos–Renyi (Erdös and Rényi 1959) and R-MAT graphs (Chakrabarti et al. 2004). In the Erdos–Renyi model, a graph is constructed by connecting vertices randomly. All edges have the same probability for existing in the graph. While Erdos–Renyi graphs consist of a uniform distribution of edges existing in the graphs, they do not model real-world behavior accurately. For that reason, we also use R-MAT graphs, since they are designed to mimic real-world graphs. An R-MAT generator creates scale-free networks designed to simulate real-world networks. This provides a basis of how we expect our method to behave on real datasets, which we also test in this section. Consider an adjacency matrix: the matrix is subdivided into four quadrants, where each quadrant has a different probability of being selected. Once a quadrant is selected, this quadrant is recursively subdivided into four subquadrants and using the same probabilities, we select one of the subquadrants. This process is repeated until we arrive at a single cell in the adjacency matrix. An edge is assigned between the two vertices making up that cell. For real-world networks, we draw from the KONECT collection of datasets (Kunegis 2013). The five datasets used are given in Table 1 and comprise a mixture of citation and social networks. These graphs are chosen because they have time stamps associated with the edges to represent temporal data.

To have a baseline for comparison, we treat scores obtained from static recomputation as ground truth. Every time we update the centrality scores using our dynamic algorithm, we recompute the centrality vector statically using Algorithm 2. Denote the vector obtained by static recomputation by $\mathbf{x}_S$ and the vector obtained by our dynamic algorithm by $\mathbf{x}_D$. We create an initial graph $G_0$ using the first half of edges, which provides a starting point for both the dynamic and static algorithms. To simulate a stream of edges in a dynamic graph, we insert the remaining edges in batches of size $b$ and apply both algorithms. For the synthetic graphs, the edges are permuted randomly during insertion. Edges in real graphs are inserted in time-stamped order. We use batch sizes of $b = 1, 10, 100$, and 1000 and vary the tolerance to which we solve for in Algorithm 1 (the Jacobi method) and provide analysis on how this affects the results of our algorithm.

### 6.2 Synthetic graphs

In this section, we present results on Erdos–Renyi and R-MAT generated graphs. For each type of graph, we generate graphs with the number of vertices as a power of 2, ranging from $2^{10}$ to $2^{14}$. We vary the average degree of the graphs from 10 to 50. For each total number of vertices and average degree, five graphs are created and tested. The results shown are averaged over these five trials. All results shown for the synthetic cases use a batch size of 1, meaning after we create the initial graph $G_0$, we sequentially insert the remaining 1/2 of edges. The trends for other batch sizes are similar.

The primary motivation behind a dynamic approach is to prune any unnecessary work in the static algorithm to develop a faster method of obtaining the centrality vector for dynamic graphs. Therefore, we evaluate the performance of the dynamic algorithm in terms of speedup compared to the static algorithm. For a particular timepoint after inserting a batch of edges, denote the time taken to compute Katz scores by the static recomputation by $T_S$ and the time taken by our dynamic algorithm as $T_D$. We calculate the algorithmic speedup in time of the dynamic algorithm against the static algorithm as

$$\text{speedup}_{\text{time}} = \frac{T_S}{T_D}.$$

Since we are using iterative methods to calculate the centrality vectors, we also evaluate the performance of the dynamic algorithm with respect to the reduction in number of iterations. For a particular timepoint $t$, denote the number of iterations taken by recomputation as $I_S$ and the time taken by the streaming approach as $I_D$. Calculate the speedup w.r.t the number of iterations as

$$\text{speedup}_{\text{iter}} = \frac{I_S}{I_D}.$$

Tables 2 and 3 give the average speedup in time and reduction in iterations, respectively, for Erdos–Renyi graphs, and Tables 4 and 5 show the same values for R-MAT graphs. As we increase the average degree for both types of graphs, the speedups in time and iterations are larger. Additionally, we see greater speedups for graphs with larger values of $n$. The dynamic algorithm likely has more of an effect for larger graphs because there is more work to be done for larger graphs with the static algorithm. Unlike the static algorithm, our dynamic algorithm only traverses parts of the graph where updates have occurred. These trends persist for both Erdos–Renyi and R-MAT graphs, but typically we find that R-MAT graphs have greater speedups than their respective Erdos–Renyi counterparts.

## 6.3 Performance on real graphs

Next we examine the performance of our algorithm on the real-world graphs. First, we look at the effect of the terminating tolerance on the speedup (in both time and iterations) obtained in Fig. 2. Specifically, Fig. 2a, b plots the speedup in time for global and personalized scores, respectively, and Fig. 2c, d plots the speedup in iterations for global and personalized scores, respectively. Results are averaged across the five real datasets and show maximum (in blue), median (in green), and minimum (in red) speedups. Note that the $y$ axis in Fig. 2a, b is on a log scale with base 10 and the $y$ axis in Fig. 2c, d is on a log scale with base 2 for clarity.

For the global scores, we observe that as the increase in the value of the tolerance to which we solve for, we obtain greater speedups. This intuitively makes sense because as we increase the value of the tolerance required to terminate (meaning a less accurate solution will suffice), the iterative solver will take fewer iterations to converge and our dynamic algorithm will have more of an effect. For the personalized scores, we see more of a plateau and the speedups obtained seem to be independent of the preset tolerance. This is due to the fact that the personalized scores themselves are so small. Therefore, it may likely take the same number of iterations to converge to a tolerance of at least $10^{-1}$ as it does to converge to $10^{-3}$ for example, so we see very little differences in the speedups for these tolerances. We also note that the speedups (in both time and iterations) for the personalized scores are greater than their global counterparts. Since the values of the scores are so small in the personalized case, the iterative solver likely takes more total iterations to converge and the dynamic algorithm has more of an effect here. Nevertheless, overall we obtain speedups of several orders of magnitude and for the global scores on average about $100\times$ speedup in time and $32\times$ speedup in iterations. Similarly for the personalized scores, we obtain on average about a $200\times$ speedup in time and about a $64\times$ speedup in iterations. Even for very low values of the tolerance (meaning we desire more precise solutions), such as $10^{-8}$, we always obtain $> 1\times$ speedup. This indicates we can obtain fairly accurate scores and with our method do so much faster than static recomputation.

Next we examine the speedups obtained as a function of batch size and compare our dynamic algorithm against two different static methods in Fig. 3. Both static methods evaluate $\mathbf{x}_S$ using Algorithm 2 but start with different initial starting vectors in line 3 in Algorithm 1 (Jacobi).

1. Method 1: uses an initial starting vector of $\mathbf{x}^{(0)} = \mathbf{0}$.
2. Method 2: uses the previous solution as a starting point for the Jacobi algorithm. Essentially, if we are computing $\mathbf{x}_{t+1}$, line 3 in Algorithm 1 becomes $\mathbf{x}^{(0)} = \mathbf{x}_t$.

Figure 3a, b plots the speedup w.r.t time versus batch size for global and personalized scores, respectively, comparing our dynamic algorithm against static recomputation. Similarly, Fig. 3c, d plots the speedup w.r.t iterations versus batch size for global and personalized scores, respectively. We show the maximum, median, and minimum speedup averaged over the 5 real graphs. Method 1 is plotted with a solid line with squares and Method 2 is plotted with a dotted line. For this,

**Table 1** Graphs used in experiments

| Graph | \|V\| | \|E\| |
|---|---|---|
| Facebook | 63,731 | 817,035 |
| Gowalla | 196,591 | 950,327 |
| DBLP | 317,080 | 1,049,866 |
| Dogster | 426,820 | 8,546,581 |
| YouTube | 1,134,890 | 2,987,624 |

Columns are graph name, number of vertices, and number of edges

**Table 2** Speedup in time for Erdos–Renyi graphs

| Average degree | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| $n = 1024$ | 1.44× | 1.62× | 1.8× | 1.99× | 2.17× |
| $n = 2048$ | 1.51× | 1.77× | 2.0× | 2.25× | 2.49× |
| $n = 4096$ | 1.66× | 2.03× | 2.37× | 2.85× | 3.34× |
| $n = 8192$ | 1.95× | 2.55× | 3.05× | 4.02× | 5.09× |
| $n = 16{,}384$ | 2.51× | 3.5× | 4.34× | 6.0× | 8.02× |

**Table 3** Speedup in iterations for Erdos–Renyi graphs

| Average degree | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| $n = 1024$ | 4.56× | 5.01× | 5.37× | 5.71× | 5.99× |
| $n = 2048$ | 4.82× | 5.4× | 5.82× | 6.17× | 6.5× |
| $n = 4096$ | 5.05× | 5.77× | 6.27× | 6.7× | 7.1× |
| $n = 8192$ | 5.25× | 6.12× | 6.69× | 7.24× | 7.73× |
| $n = 16{,}384$ | 5.40× | 6.42× | 7.04× | 7.73× | 8.33× |

**Table 4** Speedup in time for R-MAT graphs

| Average degree | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| $n = 1024$ | 1.75× | 1.95× | 2.15× | 2.44× | 2.7× |
| $n = 2048$ | 1.98× | 2.39× | 2.7× | 3.14× | 3.56× |
| $n = 4096$ | 2.42× | 3.12× | 3.62× | 4.3× | 5.08× |
| $n = 8192$ | 3.35× | 4.32× | 5.25× | 6.41× | 7.46× |
| $n = 16{,}384$ | 4.63× | 6.26× | 7.64× | 9.15× | 10.46× |

**Table 5** Speedup in iterations for R-MAT graphs

| Average degree | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| $n = 1024$ | 4.89× | 5.29× | 5.6× | 6.01× | 6.38× |
| $n = 2048$ | 5.12× | 5.77× | 6.27× | 6.73× | 7.12× |
| $n = 4096$ | 5.34× | 6.2× | 6.69× | 7.24× | 7.66× |
| $n = 8192$ | 5.81× | 6.52× | 7.18× | 7.77× | 8.25× |
| $n = 16{,}384$ | 6.0× | 6.89× | 7.62× | 8.29× | 8.72× |

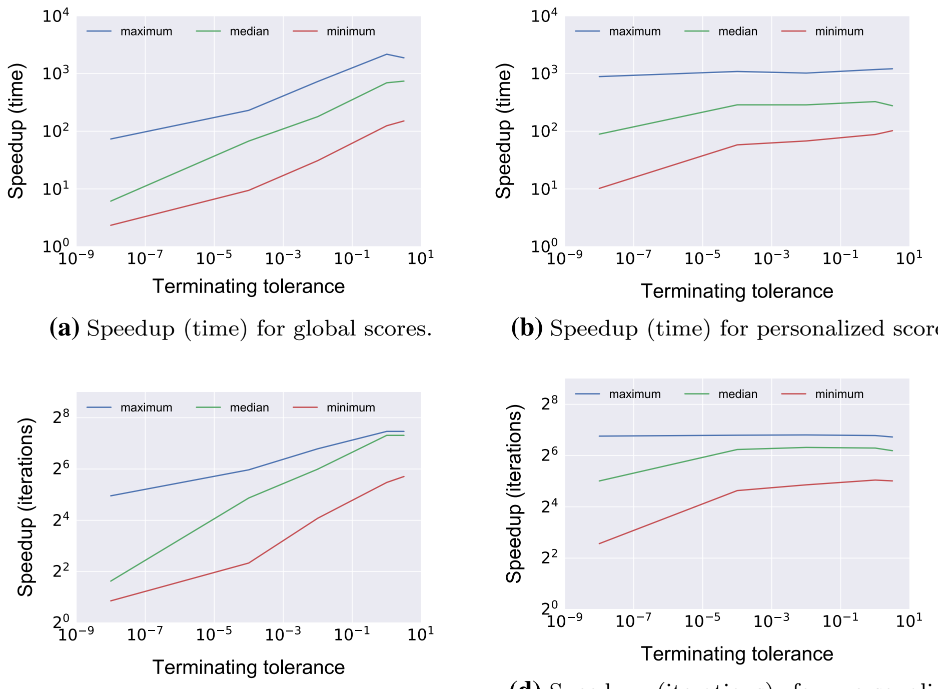
we examine results only for a terminating tolerance of $10^{-4}$ although the trends observed for other tolerances are similar. Note again that the $y$ axis in Fig. 3a, b is on a log scale with base 10 and the $y$ axis in Fig. 3c, d is on a log scale with base 2. In Fig. 3a, we see that our dynamic algorithm can be over two orders of magnitude faster for a batch size of 1 than both static recomputation approaches. It is expected that Method 2 is faster than Method 1, since we initialize Jacobi with the vector $\mathbf{x}_t$ that is likely closer to the new solution $\mathbf{x}_{t+1}$ than $\mathbf{0}$, but our dynamic algorithm is still able to outperform this method in both time and iterations. The median speedup in time for the global scores is about 100× for a batch size of 1 and about 200× for the personalized scores for a batch size of 1. Even for a batch size of 1000 edges, we always have greater than a 1× speedup. Figure 3c shows that we can obtain over an 80× reduction in iterations for both global and personalized scores for a batch size of 1. This is especially significant because the static method can take hundreds or thousands of iterations to converge in some cases, so our algorithm would provide large savings of resources in many applications. Finally, we see a greater speedup in both time and iterations for the smaller batch sizes of 1 and 10. As mentioned earlier, this is because as the batch size increases, the dynamic algorithm nears the work of a static algorithm. This shows that the dynamic approach is most useful for monitoring applications where the rankings must be updated after only a small number of data changes.

Next we examine the behavior of both algorithms with respect to raw iteration counts over time. Henceforth when referring to the static algorithm, we use Method 2 from above. Figure 4 plots the raw number of iterations used by the static (the dotted blue line) and dynamic (the solid green line) algorithms for different batch sizes for the Facebook graph. We sample at 100 evenly spaced timepoints for each batch size. Figure 4a–d plots the comparison for batch sizes $b = 1, 10, 100, 1000$, respectively. All four figures show the same general behavior: while the number of iterations for static recomputation continues to steadily increase as edges are added into the graph, the dynamic algorithm maintains a stable number of iterations over time. This is because the dynamic algorithm only targets the places in the vector that are affected by edge updates. For example, take Fig. 4b. The dotted blue line shows that the number of iterations for the static recomputation of the centrality vector continually increases over time as more edges are added into the graph, eventually reaching about 175 iterations once all edges are added. However, for the dynamic algorithm shown in the solid green line, the number of iterations is stable at around 1–20 iterations for all points in time. It is important to note that this trend persists regardless of the batch size. Even for very large batch sizes of $b = 1000$, while there are small fluctuations in the number of iterations, there is no trend of increasing iteration counts over time, meaning our algorithm is robust to many edge insertions.

## 6.4 Quality on real graphs

We have seen that we are able to achieve results faster using a dynamic algorithm compared to static recomputation every time the graph changes when calculating centrality scores in dynamic networks. However, it is also important to ensure that the centrality scores returned by the dynamic algorithm are similar to those returned by the static algorithm. To evaluate the quality of our algorithm, we measure two quantities: (1) recall of top $k$ vertices measured as

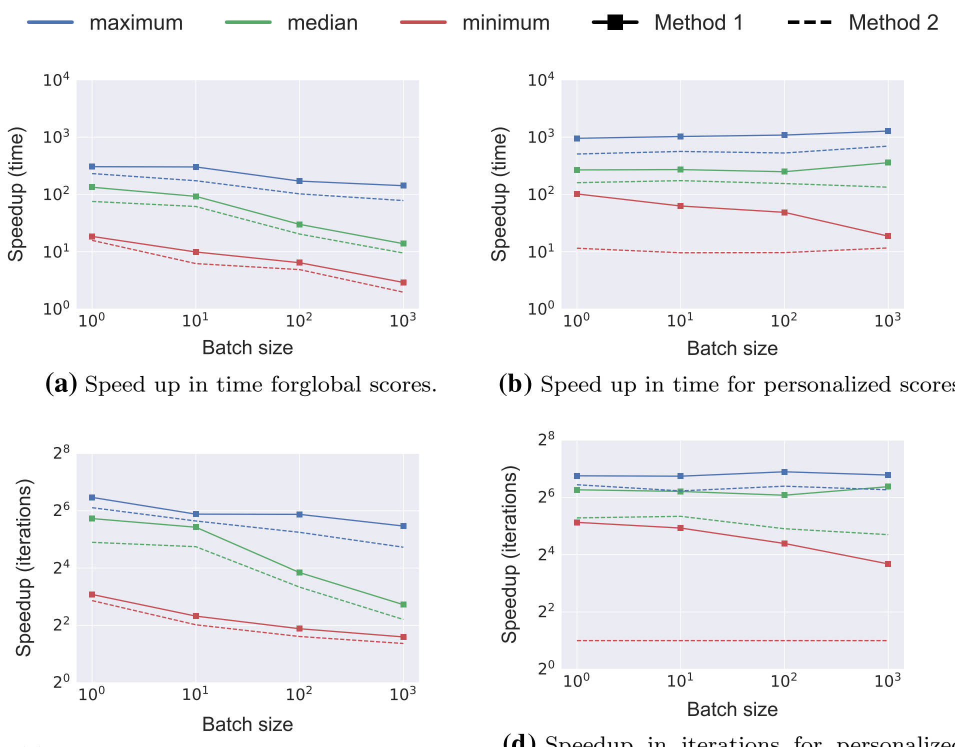
**Fig. 2** Speedup (time) for global scores. Higher is better. **a** Speedup (time) for personalized scores, **b** Speedup (iterations) for global scores, **c** Speedup (iterations) for personalized scores, **d** Speedup (time and iterations) versus tolerance



**(a)** Speedup (time) for global scores. **(b)** Speedup (time) for personalized scores

**(c)** Speedup (iterations) for global scores. **(d)** Speedup (iterations) for personalized scores.

**Fig. 3** Speedup in time for global scores, **a** speedup in time for personalized scores, **b** speedup in iterations for global scores, **c** speedup in iterations for personalized scores, **d** speedup (time and iterations) versus batch size. Higher is better



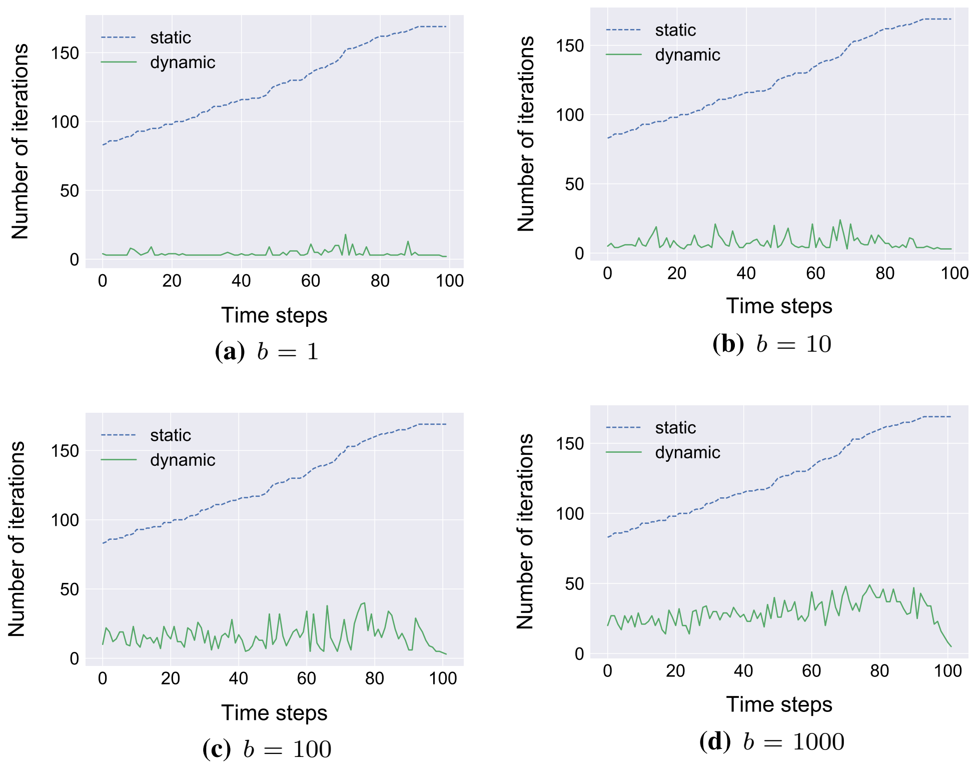
**(a)** Speed up in time forglobal scores. **(b)** Speed up in time for personalized scores.

**(c)** Speed up in iterations for global scores. **(d)** Speedup in iterations for personalized scores.

$$\text{recall}_k = \frac{|C_S(k) \cap C_D(k)|}{|C_S(k)|},$$

where $C_S(k)$ and $C_D(k)$ are the set of the top $k$ highly ranked vertices from the statically and dynamically computed centrality vectors, respectively, and (2) average error computed

**Fig. 4** Raw number of iterations for the FACEBOOK graph for different batch sizes. Dynamic algorithm is plotted in solid green line and static algorithm is plotted in dotted blue line. **a** $b = 1$, **b** $b = 10$, **c** $b = 100$, **d** $b = 1000$



as the pointwise difference between the statically and dynamically computed vectors

$\text{error} = \|\mathbf{x}_S - \mathbf{x}_D\|_\infty$.

Table 6 presents the average recall of the top 10, 100, and 1000 vertices in the different graphs for both our dynamic algorithm and the alternate approach presented in Sect. 4.3. We use a terminating tolerance of $10^{-4}$. Immediately we note that our algorithm has a perfect recall of the top $k$ vertices in all cases except for one graph (DBLP) for one value of $k = 100$, and the recall is 0.99 here. The quality of the alternate approach suffers and is not able to maintain perfect recall in many cases. Furthermore, will see next that the actual values of the scores themselves (measured by the average error) between the dynamically computed vector from the alternate method compared to static recomputation are not similar at all, and we obtain very high errors using this alternate method.

Table 7 presents the average error for each of the graphs tested and for all batch sizes for both our dynamic method and the alternate method. We again use results from a tolerance of $10^{-4}$. The average error obtained from our dynamic algorithm for global and personalized scores is 1.32e−02 and 8.69e−05, respectively. However, the average error obtained from the alternate approach compared to static recomputation for global and personalized scores is 6.19e+03 and 1.14e−01, respectively. For both global and personalized scores, the errors from the alternate method are several

**Table 6** Summary statistics of recall of top vertices for different graphs for a terminating tolerance of $10^{-4}$

| Type | Graph | Top 10 | Top 100 | Top 1000 |
|---|---|---|---|---|
| (a) Our dynamic algorithm | | | | |
| Global | Facebook | 1.00 | 1.00 | 1.00 |
| | Gowalla | 1.00 | 1.00 | 1.00 |
| | DBLP | 1.00 | 0.99 | 1.00 |
| | Dogster | 1.00 | 1.00 | 1.00 |
| | YouTube | 1.00 | 1.00 | 1.00 |
| Personalized | Facebook | 1.00 | 1.00 | 1.00 |
| | Gowalla | 1.00 | 1.00 | 1.00 |
| | DBLP | 1.00 | 1.00 | 1.00 |
| | Dogster | 1.00 | 1.00 | 1.00 |
| | YouTube | 1.00 | 1.00 | 1.00 |
| (b) Alternate approach | | | | |
| Global | Facebook | 0.91 | 0.84 | 0.89 |
| | Gowalla | 0.92 | 1.00 | 0.99 |
| | DBLP | 1.00 | 0.93 | 0.92 |
| | Dogster | 1.00 | 0.95 | 0.96 |
| | YouTube | 1.00 | 0.97 | 0.95 |
| Personalized | Facebook | 0.89 | 0.94 | 0.91 |
| | Gowalla | 0.90 | 0.93 | 0.96 |
| | DBLP | 0.95 | 0.97 | 0.95 |
| | Dogster | 0.98 | 0.92 | 0.91 |
| | YouTube | 0.93 | 0.82 | 0.87 |

**Table 7** Summary statistics of average error versus batch size for different graphs for a terminating tolerance of $10^{-4}$

| Type | Graph | $b = 1$ | $b = 10$ | $b = 100$ | $b = 1000$ |
|---|---|---|---|---|---|
| (a) Our dynamic algorithm | | | | | |
| Global | Facebook | 1.64e−03 | 2.77e−03 | 4.52e−03 | 5.00e−03 |
| | Gowalla | 6.52e−03 | 1.55e−02 | 2.38e−02 | 2.95e−02 |
| | DBLP | 3.32e−05 | 9.87e−05 | 2.88e−04 | 1.89e−03 |
| | Dogster | 2.01e−03 | 1.75e−02 | 2.05e−02 | 2.01e−02 |
| | YouTube | 7.78e−03 | 2.17e−02 | 3.67e−02 | 4.58e−02 |
| Personalized | Facebook | 6.11e−07 | 2.76e−06 | 1.71e−05 | 1.08e−03 |
| | Gowalla | 5.11e−07 | 2.51e−06 | 3.54e−04 | 2.41e−04 |
| | DBLP | 6.03e−09 | 7.53e−09 | 7.20e−09 | 1.29e−05 |
| | Dogster | 1.08e−07 | 2.13e−06 | 4.48e−06 | 1.23e−05 |
| | YouTube | 1.34e−07 | 3.36e−06 | 1.11e−06 | 5.38e−06 |
| (b) Alternate approach | | | | | |
| Global | Facebook | 1.84e+03 | 1.84e+03 | 1.84e+03 | 1.84e+03 |
| | Gowalla | 2.93e+03 | 2.93e+03 | 2.93e+03 | 2.92e+03 |
| | DBLP | 6.15e+01 | 6.15e+01 | 6.14e+01 | 6.14e+01 |
| | Dogster | 2.20e+03 | 8.59e+03 | 2.61e+04 | 2.74e+04 |
| | YouTube | 8.03e+03 | 1.08e+04 | 1.08e+04 | 1.08e+04 |
| Personalized | Facebook | 8.48e−03 | 4.22e−03 | 6.73e−01 | 7.07e−01 |
| | Gowalla | 4.56e−02 | 8.91e−02 | 1.05e−02 | 4.15e−03 |
| | DBLP | 1.18e−03 | 4.40e−05 | 4.57e−02 | 8.40e−05 |
| | Dogster | 4.33e−02 | 4.03e−02 | 1.01e−02 | 3.79e−01 |
| | YouTube | 1.07e−01 | 1.54e−02 | 3.25e−02 | 5.87e−02 |

orders of magnitude higher than the corresponding errors from our method. In fact, the errors for the global scores from the alternate method are in the thousands or tens of thousands. The errors for the personalized scores from the alternate method are significantly smaller than the errors for the global scores from the alternate method (on the order of $\approx 10^{-2}$). However, the values in the personalized centrality vector themselves are on the order of $10^{-2}$ to $10^{-3}$ so errors of $\approx 10^{-2}$ for the personalized scores from the alternate approach still indicate that this is a poor method.

Next we look at the behavior of both the alternate method and our dynamic algorithm over time. Figure 5 plots the average error over time for our dynamic algorithm (the figures on the left) and the alternate method (the figures on the right). We show results for a batch size of 1 for global scores, although results for other batch sizes are similar. For our dynamic algorithm, we note that for no graph do we see a trend of error increasing over time, unlike those from the alternate method. In fact using our dynamic algorithm, the average error in the two largest graphs (Gowalla and YouTube) actually decreases as we insert more edges into the graph using our dynamic algorithm. This is in stark contrast to the alternate method where we see only a trend of error increasing over time showing that the forward error analysis approach only accumulates error instead of converging to the answer obtained by static recomputation. Additionally, note that the scales of the $y$ axis on the figures plotting results from our dynamic algorithm are at most $10^{-2}$ indicating that values in the vector obtained from our dynamic algorithm match those obtained from static recomputation, while the $y$ axis on the figures from the alternate method range as high as $10^4$. In summary, we note that the alternate method presented is not sufficient to calculate the updated centrality metric and the increasing and large values of the average error prove this method returns results of poor quality.

Finally, Fig. 6 explores in detail the underlying impact of the time step granularity on the quality of our algorithm. Figure 6a plots the error versus batch size for the global scores and Fig. 6b plots the error versus batch size for the personalized scores for all five real graphs tested. In both cases, we see a trend of increasing error as a function of increasing batch size. This is because the underlying assumption of our algorithm relies on the fact that there exists smoothness between consecutive time steps. With a larger number of edge insertions in one batch, the solutions before and after the batch of insertions will differ considerably. Therefore, it is not surprising that larger batch sizes impact the quality of the algorithm more than smaller batch sizes. However, even though there is a trend of increasing error for larger batch sizes compared to smaller batch sizes, the average error is still relatively low compared to the values in the centrality vector themselves, and we therefore conclude that our dynamic algorithm is able to maintain similar quality to static recomputation.

## 7 Adding and removing vertices

Adding and removing edges is fairly straightforward since edges only require updating the $\Delta A$ matrix with either a 1 (insertions) or −1 (deletions) in the corresponding position for the edge in question. However, adding and removing vertices becomes slightly trickier, since our work is based in linear algebra with fixed size matrices. One solution to this is to assume some reasonable bound on the total number of vertices allowed (this can be application dependent or based on available storage). The algorithm would then start with a matrix $A_0$ with empty rows for vertices that do not exist yet in the graph and as the vertices are added with edges into the existing graph, the corresponding rows are also updated. Deleting a vertex can be handled in a similar manner by allowing the vertex to technically exist but remain disconnected from the entire graph. Essentially when deleting vertex $i$ from the graph, we can cope by zeroing out the $i$th row in the adjacency matrix.

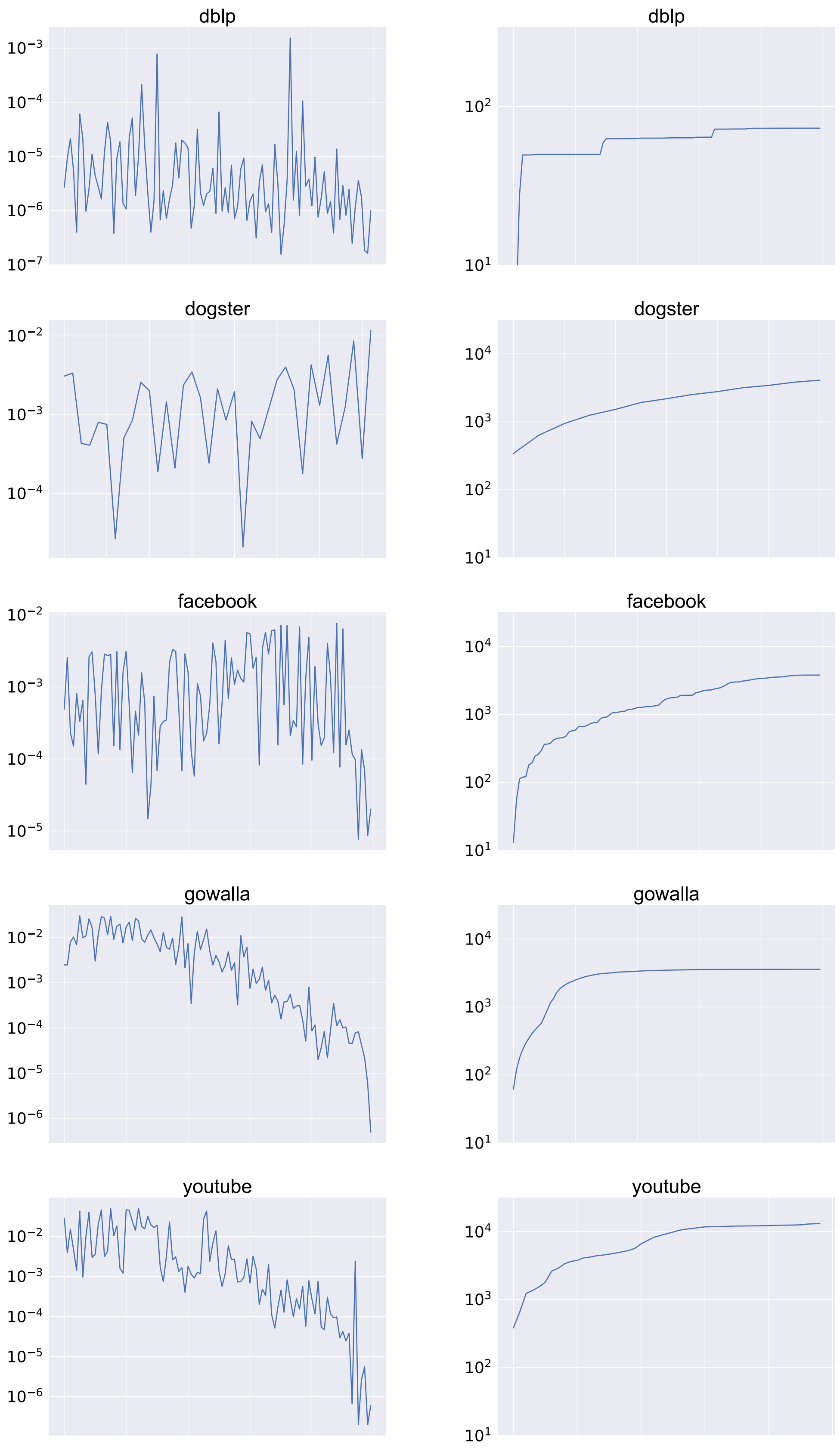
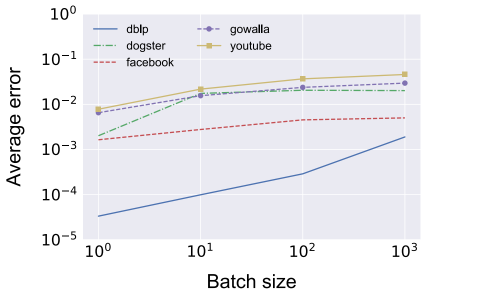
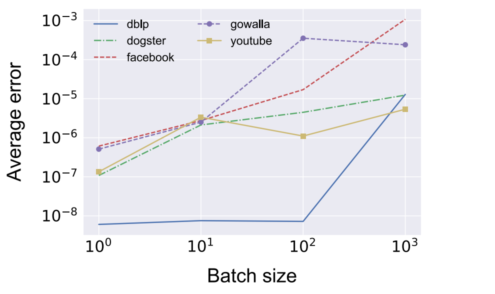


**Fig. 5** Average error plotted over time for both our dynamic algorithm (left figures) and the alternate method (right figures). Results are shown for a batch size of 1 and for global scores. Lower values are better

**Fig. 6** Effect of time step granularity (batch size of edge insertions) on quality of our algorithm. **a** Average error versus batch size for global scores, **b** Average error versus batch size for personalized scores



**(a)** Average error versus batch size forglo bal scores.



**(b)** Average error versus batch size for personalized scores.

## 8 Conclusions

We have presented a new algorithm that incrementally updates the Katz centrality scores when the underlying graph changes. Our dynamic algorithm is faster than statically recomputing the centrality scores every time the graph changes, and the performance improvement is greatest when low-latency updates are required. However, our approach is still faster than recomputing from scratch even for large batch insertions of edges into the graph. We compared our method to a static recomputation initialized from the all zeros vector and from the previous time step's solution and showed that our method is able to outperform both. Our dynamic algorithm returns scores that are within negligible error of the scores returned by static recomputation, and we showed that the quality of the scores using our dynamic algorithm does not deteriorate over time. We presented and explained the problems associated with a simple intuitive iterative approach and compared it to our dynamic algorithm and showed that our method is far superior and is able to maintain good quality of results and does not accumulate error over time, unlike the alternate method. We analyzed the effect of the time step granularity on the quality of our dynamic algorithm and showed that even though the error between the results of our method and static recomputation increases for larger batch sizes, the overall error is still relatively small compared to the actual values of the centrality scores themselves and is therefore negligible. Moreover, our algorithm returns perfect recall of top vertices across all graphs in nearly all cases. Finally, we presented an approach for dealing with the addition and removal of vertices from a dynamic graph, which may be addressed in future work.

**Acknowledgements** This work is in part supported by a graduate fellowship from the National Physical Science Consortium. The work depicted in this paper was sponsored in part by the National Science Foundation under awards #1339745 and #1535058. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

## References

Albert R, Jeong H, Barabási AL (1999) Internet: diameter of the world-wide web. Nature 401(6749):130–131

Bahmani B, Chowdhury A, Goel A (2010) Fast incremental and personalized pagerank. Proc VLDB Endow 4(3):173–184

Bavelas A (1950) Communication patterns in task-oriented groups. J Acoust Soc Am 22:723–730

Benzi M, Estrada E, Klymko C (2013) Ranking hubs and authorities using matrix functions. Linear Algebra Appl 438(5):2447–2474

Benzi, M, Klymko C (2014) A matrix analysis of different centrality measures. arXiv preprint arXiv:1312.6722

Chakrabarti D, Zhan Y, Faloutsos C (2004) R-mat: a recursive model for graph mining. In: SDM, vol 4. SIAM, pp 442–446

Chen YY, Gan Q, Suel T (2004) Local methods for estimating PageRank values. In: Proceedings of the thirteenth ACM international conference on Information and knowledge management. ACM, pp 381–389

Chien S, Dwork C, Kumar R, Sivakumar D (2001) Towards exploiting link evolution

Erdös P, Rényi A (1959) On random graphs, I. Publicationes Mathematicae (Debrecen) 6:290–297

Freeman LC (1977) A set of measures of centrality based on betweenness. Sociometry 40(1):35–41

Gleich DF (2015) Pagerank beyond the web. SIAM Rev 57(3):321–363

Green O, McColl R, Bader DA (2012) A fast algorithm for streaming between ness centrality. In: Privacy, security, risk and trust (PASSAT), 2012 international conference on and 2012 international conference on social computing (SocialCom). IEEE, pp 11–20

Gyöngyi Z, Garcia-Molina H, Pedersen J (2004) Combating web spam with trust rank. In: Proceedings of the thirtieth international conference on very large data bases, vol 30. VLDB Endowment, pp 576–587

Higham NJ (2002) Accuracy and stability of numerical algorithms. SIAM

Higham NJ (2008) Functions of matrices: theory and computation. SIAM

Katz L (1953) A new status index derived from sociometric analysis. Psychometrika 18(1):39–43

Kempe D, Kleinberg J, Tardos É (2003) Maximizing the spread of influence through a social network. In: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, pp 137–146

Kokiopoulou E, Chen J, Saad Y (2011) Trace optimization and eigenproblems in dimension reduction methods. Numer Linear Algebra Appl 18(3):565–602

Kunegis J (2013) Konect: the koblenz network collection. In: Proceedings of the 22nd international conference on World Wide Web. ACM, pp 1343–1350

Langville AN, Meyer CD (2002) Updating PageRank using the group inverse and stochastic complementation. Informe técnico crsc02-tr32

Langville AN, Meyer CD (2004) Updating the stationary vector of an irreducible Markov chain with an eye on Googles PageRank. SIMAX, Citeseer

Nathan E, Bader DA (2017) A dynamic algorithm for updating katz centrality in graphs. In: Proceedings of the 2017 IEEE/ACM international conference on advances in social networks analysis and mining, Sydney, Australia, vol 31

Page L, Brin S, Motwani R, Winograd T (1999) The PageRank citation ranking: bringing order to the web. Technical Report, Stanford InfoLab

Riedy J (2016) Updating pagerank for streaming graphs. In: IEEE international parallel and distributed processing symposium workshops, 2016, pp 877–884

Saad Y (2003) Iterative methods for sparse linear systems. SIAM

Sariyuce AE, Kaya K, Saule E, Catalyurek UV (2013) Incremental algorithms for closeness centrality. In: IEEE international conference on big data, 2013, pp 487–492

Sarma AD, Gollapudi S, Panigrahy R (2011) Estimating PageRank on graph streams. JACM 58(3):13

Wei W, Carley K (2014) Real time closeness and betweenness centrality calculations on streaming network data. Academy of Science and Engineering, Los Angeles, USA

Wilkinson JH (1994) Rounding errors in algebraic processes. Courier Corporation