Approximating Personalized Katz Centrality in Dynamic Graphs

Eisha Nathan(✉) and David A. Bader

School of Computational Science and Engineering,
Georgia Institute of Technology, Atlanta, GA 30363, USA
enathan3@gatech.edu, bader@cc.gatech.edu

Abstract. Dynamic graphs can capture changing relationships in many real datasets that evolve over time. One of the most basic questions about networks is the identification of the "most important" vertices in a network. Measures of vertex importance called centrality measures are used to rank vertices in a graph. In this work, we focus on Katz Centrality. Typically, scores are calculated through linear algebra but in this paper we present an new alternative, agglomerative method of calculating Katz scores and extend it for dynamic graphs. We show that our static algorithm is several orders of magnitude faster than the typical linear algebra approach while maintaining good quality of the scores. Furthermore, our dynamic graph algorithm is faster than pure static recomputation every time the graph changes and maintains high recall of the highly ranked vertices on both synthetic and real graphs.

Keywords: Katz Centrality · Dynamic graphs
Approximate centrality · Personalized centrality

1 Introduction

Graphs are used to represent relationships between entities, whether in web traffic, financial transactions, or society [1]. In real-world networks, new data is constantly being produced, leading to the notion of dynamic graphs. The identification of central vertices in an evolving network is a fundamental problem in network analysis [2]. Centrality measures provide a score for each vertex in the graph and the scores can be turned into rankings on the vertices. While in many applications, centrality scores are used to measure global importance in the entire network, there are also several applications that require the use of personalized centrality scores, or scores calculated with respect to specific seed vertices of interest. Consider performing a web search in Google. Typically the user desires a set of webpages most relevant to a specific search query (or personalized w.r.t. the search query), not a set of the most highly visited pages in general. However, as the size of the network increases and more and more data gets added to the graph, calculating exact centrality scores becomes increasingly computationally intensive, and we therefore seek alternative methods of estimating the scores.

© Springer International Publishing AG, part of Springer Nature 2018
R. Wyrzykowski et al. (Eds.): PPAM 2017, LNCS 10777, pp. 290–302, 2018.
https://doi.org/10.1007/978-3-319-78024-5_26

In this paper we focus on Katz Centrality, a centrality metric that measures the affinity between vertices as a weighted sum of the number of walks between them [3]. We present a new algorithm for approximating personalized Katz Centrality scores and extend our algorithm for use on dynamic graphs.

1.1 Contributions

We present a new algorithm for approximating personalized Katz Centrality (Static_Katz) and extend our algorithm for dynamic graphs (Dynamic_Katz). We show Static_Katz provides good quality approximations and is several orders of magnitude faster when compared to the conventional linear algebraic method of computing Katz scores. Dynamic_Katz is faster when compared to a pure static recomputation and preserves the ranking of vertices in evolving networks. We present results on both synthetic and real-world graphs.

1.2 Related Work

There exist many centrality measures in the literature to calculate vertex importance. Betwenness centrality is a very popular metric where a high betweenness centrality for a vertex indicates that removal of this vertex will cause a large number of shortest paths to not exist anymore in the network [4]. An incremental algorithm to update betweenness centrality values in dynamic graphs by maintaining additional data structures to store previously computed values is proposed in [5]. PageRank is the most similar metric to Katz Centrality and was first introduced to rank webpages in a web search [6]. Vertices with a high PageRank scores indicate that random walks through the graph tend to visit these vertices. The authors in [7] analyze the efficiency of Monte Carlo methods for incremental computation of PageRank in dynamic graphs by maintaining a small number of short random walk segments starting at each vertex in the graph, and are able to provide highly accurate estimations of the values for the top R vertices. [8] proposes a method to update the eigenvalue formulation of PageRank to update the corresponding ranking vector. They use the power method to do so but the method eventually ends up requiring access to the entire graph which becomes very memory intensive. In [9], an algorithm for updating PageRank values in dynamic graphs through sparse updates to the residual is presented.

In this work we develop agglomerative algorithms to compute Katz Centrality. Typically Katz Centrality scores are calculated using linear algebraic computations. There has been some prior work in approximating Katz scores in static graphs using linear algebraic techniques. Several methods have only examined walks up to a certain length [10] or employ low-rank approximation [11]. However, as far as the authors are aware, there is no prior work in developing a dynamic agglomerative algorithm for updating Katz scores in graphs.

The main contribution of this paper is the development of new agglomerative algorithms for calculating approximate personalized Katz scores in static and dynamic graphs. We present our algorithms in Sect. 2. Section 3 evaluates our methods with respect to performance and quality, and in Sect. 4 we conclude.

2 Algorithms

Here we present our static (STATIC_KATZ) and dynamic (DYNAMIC_KATZ) algorithm for computing personalized Katz scores of the vertices in a graph.

2.1 Definitions

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 $A(i, j) = 1$ if there is an edge $(i, j) \in E$ and $A(i, j) = 0$ otherwise. We use undirected, unweighted graphs so $\forall i, j, A(i, j) = A(j, i)$ and all edge weights are 1. A dynamic graph changes 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 and denote the current snapshot of the dynamic graph G at time t by $G_t = (V_t, E_t)$. In this work, we focus only on edge insertions to the graph, and the vertex set stays the same over time so $\forall t, V_t = V$.

Katz Centrality scores ($\mathbf{c}$) count the number of weighted walks in a graph starting at vertex i, penalizing longer walks with a user-chosen parameter α, where $\alpha \in (0, 1/\|A\|_2)$ and $\|A\|_2$ is the matrix 2-norm of A. A walk in a graph traverses edges between a series of vertices $v_1, v_2, \cdots, v_k$, where vertices and edges are allowed to repeat. It is a well-known fact in graph theory that powers of the adjacency matrix represent walks of different lengths between vertices in the graph. Specifically, $A^k(i, j)$ gives the number of walks of length k from vertex i to vertex j [12]. To count weighted walks of different lengths in the graph, we can sum powers of the adjacency matrix using the infinite series

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

Provided α is chosen to be within the appropriate range, this infinite series converges to the matrix resolvent $(I - \alpha A)^{-1}$. When Katz Centrality was first introduced in 1953, Katz used the row sums to calculate vertex importance to obtain centrality scores as $(I - \alpha A)^{-1}\mathbf{1}$, where $\mathbf{1}$ is the $n \times 1$ vector of all 1s. These are referred to as *global Katz scores* and count the total sum of the number of weighted walks of different length starting at each vertex. We extrapolate from this definition *personalized Katz scores*, where the ith column of the matrix $(I - \alpha A)^{-1}$ represents the personalized scores with respect to vertex i, or the weighted counts of the number of walks from vertex i to all other vertices in the graph. Mathematically, we can write the personalized Katz scores with respect to vertex i as $(I - \alpha A)^{-1}\mathbf{e}_i$, where $\mathbf{e}_i$ is the ith canonical basis vector, the vector of all 0s except a 1 in the ith position. We set $\alpha = 0.85/\|A\|_2$ as in [13].

Typically Katz Centrality scores are calculated using linear algebra by solving the linear system $\mathbf{c} = (I - \alpha A)^{-1}\mathbf{1}$ for the global scores or $\mathbf{c} = (I - \alpha A)^{-1}\mathbf{e}_i$ for the personalized scores [14]. If the system is fairly small (meaning the $n \times n$ matrix $I - \alpha A$ is not very large), we can solve for $\mathbf{c}^*$ exactly in $\mathcal{O}(n^2)$ using Cholesky decomposition. In many cases since n may be very large, we use iterative solvers

to obtain an approximation in optimally $\mathcal{O}(m)$ time, provided the number of iterations is not very large. An iterative method approximates the solution $\mathbf{x}$ in a linear system $M\mathbf{x} = \mathbf{b}$ given a matrix M and vector $\mathbf{b}$ by starting with an initial guess $\mathbf{x}_0$ and iteratively improving the current guess until reaching some sort of terminating criterion [15]. Usually this criterion is based off of correctness of the current guess $\hat{\mathbf{x}}$ and the iterative solver terminates when $\|M\hat{\mathbf{x}} - \mathbf{b}\|_2 < tol$, where tol is some predetermined tolerance (usually $\approx 10^{-15}$). By setting up a linear system for Katz scores as $M\mathbf{c} = \mathbf{b}$ where $\mathbf{b}$ is either $\mathbf{1}$ (global) or $\mathbf{e}_i$ (personalized) and $M = I - \alpha A$, we can use iterative methods to solve for $\mathbf{c}$.

While solving the linear system works fairly well for the global scores, in the personalized case many of the vertices have scores close to 0 if they are very far away from the seed vertex i. Therefore, solving the linear system above for personalized scores becomes increasingly computationally intensive because it requires many iterations to converge. For this reason, in this paper we present an agglomerative algorithm as an alternative to the typical linear algebra approach to calculate approximate personalized Katz scores. We calculate scores by examining the actual network structure itself to count walks without using linear algebra. Our algorithm assumes a single seed vertex but can be extended to allow for multiple seed vertices. Henceforth, we use *seed* to denote the seed vertex (so we are computing personalized Katz scores with respect to vertex *seed*).

2.2 Static Algorithm

Since walks in graphs allow for repeats of vertices and edges, calculating exact Katz Centrality scores involves counting walks up til infinite lengths. In practice this is not feasible and so the algorithm we present calculates only approximate Katz Centrality scores. To approximate scores, we count walks only up to length k. We denote the vector of personalized Katz scores obtained by only counting walks up til length k w.r.t. *seed* as $\mathbf{c}_k = (I + \alpha A + \alpha^2 A^2 + \cdots + \alpha^k A^k)\mathbf{e}_{seed}$.

In Static_Katz, we maintain three separate data structures:

- an $n \times k$ array *walks* to count the number of walks in the graph. The (i, j)th entry in this array indicates how many walks of length j exist from *seed* to vertex i.
- a queue *map* to indicate what vertices are reachable at the current iteration. At each iteration j, the value of $map[vtx]$ indicates how many walks of length j exist from *seed* to vertex vtx.
- an $n \times 1$ array *visited*, where $visited[i]$ gives the iteration at which vertex i was initially reached from *seed*. This array is primarily used in our dynamic algorithm.

The overarching static algorithm is given in Algorithm 1 and is split into two subroutines. The first subroutine in Algorithm 2, Compute_Walks, counts the number of walks. To do so, we implement a variant of breadth-first search. The queue *map* is initialized with the source vertex *seed*. At each iteration j, we perform the following main steps:

1. Iterate through all vertices v in *map* (line 7)
2. If we haven't already visited vertex v, we set the value of $visited[v]$ to the current iteration j (line 9)
3. This is the key step in calculating the number of walks. Here, $N(v)$ indicates the set of neighbors of vertex v. For each neighbor vertex, we propagate the number of walks from v. If there are *count* number of walks from *seed* to v of length $j-1$, then for each neighbor *dest* of v, there are *count* number of walks from *seed* to *dest* of length j going through v (line 11)
4. Finally, we set the values in the *walks* array for the current iteration j to indicate how many total number of walks are possible from the source vertex *seed* to all vertices reachable in the current iteration (line 13)

The second subroutine in Algorithm 3, Calculate_Scores, calculates the personalized Katz scores using the *walks* array. The Katz score for vertex i is the weighted (by powers of α) sum of walks of all lengths up to k from *seed* to i.

Algorithm 1. Static algorithm to compute Katz scores from source vertex *seed* up to walks of length k.

```
1: procedure STATIC_KATZ(G, seed, k, α)
2:     walks = COMPUTE_WALKS(G,seed,k)
3:     c = CALCULATE_SCORES(walks,α)
4: return c
```

Algorithm 2. Static algorithm to recompute counts of walks up to length k from source vertex *seed*.

```
1: procedure COMPUTE_WALKS(G, seed, k)
2:     walks = n × k array initialized to 0
3:     visited = n × 1 array initialized to -1
4:     map[seed] = 1                              ▷ Initialize map
5:     j = 0
6:     while j < k do
7:         for v in map do
8:             count = map[v]
9:             if visited[v]==-1 then
10:                visited[v] = j
11:            for nbr in N(v) do
12:                map[nbr] += count
13:        for v in map do           ▷ Count walks of length j in current iteration
14:            walks[v][j] = map[v]
15:        j += 1
       return walks
```

Algorithm 3. Calculate Katz scores from walk counts.

```
1: procedure CALCULATE_SCORES(walks, α)
2:     c = n × 1 array initialized to 0
3:     for i = 1 : n do
4:         for j = 1 : k do
5:             c[i] += α^{j+1} · walks[i][k]
       return c
```

Denote the result of STATIC_KATZ as $\mathbf{c}_k$ and the exact solution (obtained through linear algebra) as $\mathbf{c}_*$. We can bound the pointwise error between our approximation $\mathbf{c}_k$ and the exact solution $\mathbf{c}^*$ by ϵ_k as follows:

$$
\begin{aligned}
\|\mathbf{c}^* - \mathbf{c}_k\|_\infty &= \|\sum_{p=0}^{\infty} \alpha^p A^p - \sum_{p=0}^{k} \alpha^p A^p\|_2 \\
&= \|\sum_{p=k+1}^{\infty} \alpha^p A^p\|_2 \\
&= \|\alpha^{k+1} A^{k+1} \sum_{p=0}^{\infty} \alpha^p A^p\|_2 \\
&\leq |\alpha^{k+1}| \|A^{k+1}\|_2 \|I - \alpha A)^{-1}\|_2 \\
&\leq \alpha^{k+1} \frac{\|A\|_2^{k+1}}{\lambda_{min}(I - \alpha A)} := \epsilon_k s
\end{aligned}
$$

Note that this proof means that the scores provided from our approximation will never be greater than ϵ_k away from the exact scores. We will see in Sect. 3 that this bound not only provides reasonable results but our approximation empirically produces scores also several orders of magnitude closer than what is theoretically guaranteed.

While results in Sect. 3 only examine starting at a single seed vertex, our algorithm can easily be adapted to the case where we allow multiple seed vertices. Instead of initializing the *map* with only the single seed vertex in Line 4 in Algorithm 2, we simply initialize the map with all desired seed vertices. The rest of the algorithm can remain the same as we will then count walks from all seed vertices. The complexity of our static algorithm is $\mathcal{O}(km)$. This is because at each iteration we can touch at most m edges and we run our algorithm a total of k times to count walks up to length k.

2.3 Dynamic Algorithm

The overall dynamic algorithm DYNAMIC_KATZ for updating personalized Katz scores is given Algorithm 4 and uses a helper function UPDATE_WALKS, described in Algorithm 5. For our dynamic algorithm we consider the case where we insert a single edge e into the graph between vertices *src* and *dest*. Instead of a complete static recomputation, we can avoid unnecessary computation by using the previously described *visited* array. If we insert an edge between vertices *src* and

dest, we only need to update counts of walks for vertices that have been visited after vertices *src* and *dest*. Furthermore, we only need to update counts for walks that use the newly added edge. Given a starting vertex *curr_vtx* and integer j, the function UPDATE_WALKS propagates the updated counts of walks from *curr_vtx* to the remaining vertices starting at walks of length j. We do this by maintaining a queue of walk counts for each vertex visited using a variant of breadth-first search, similar to the static algorithm described earlier. The key step is in line 8, where we only traverse walks and update the walk count if we are using the newly added edge. This effectively prunes the amount of work done compared to a pure static recomputation.

In Algorithm 4, DYNAMIC_KATZ, for an inserted edge $e = (src, dest)$ we calculate which vertex has been visited first (lines 2–6). Without loss of generality, suppose *src* had originally been visited first. In line 7, we update the *visited* value of *dest* because we can now get to *dest* from *src* using the newly added edge. Accordingly, we increment by one the number of walks possible for *dest* as a direct result of the new edge in line 8. For the inserted edge e, the function DYNAMIC_KATZ calls the helper function UPDATE_WALKS for both affected vertices *src* and *dest* to update the walk counts. For vertex *src*, we start updating walks of length $visited[src] + 1$ and similarly for vertex *dest* for walks of length $visited[dest] + 1$. Adding these updated counts to the existing array *walks* effectively propagates the effect of adding the new edge and then in line 11 we calculate the updated Katz scores. Once we have the updated walks, we can calculate the scores using Algorithm 3 as we did in the static recomputation.

Note that our dynamic algorithm is an approximation to the static recomputation. While updating the walk counts for *src* and *dest* using the new edge accounts for much of the effect of the added edge, it is possible there are walks originating from other vertices in the network that go through the added edge that need to be updated. However, the effect of these extra walks will be minimal compared to the effect from the *src* and *dest* vertices, and we show that our dynamic algorithm maintains good quality compared to a static recomputation when concerned about recall of the highly ranked vertices in Sect. 3. The worst-case complexity of our dynamic algorithm is still the same as the static algorithm, $\mathcal{O}(km)$, because in the worst-case we can still have to touch m edges at each iteration. However empirically we see that we still obtain significant speedups compared to the static algorithm in Sect. 3 because in practice our dynamic algorithm only traverses an edge if the walk in question uses the newly added edge.

We illustrate our dynamic algorithm on a small toy network. Figure 1 depicts the initial graph and the corresponding walk counts of length k up til $k = 3$ for $seed = 0$. In Fig. 2, we add an edge between vertices 2 and 5 and show the updated walk counts desired in red. The *visited* array is updated accordingly, since we can now reach vertex 5 through vertex 2. When we update the walk counts from vertex 5 starting at walks of length $visited[5] + 1 = 3$, we obtain a new walk of length 3 to vertex 2 that uses the new edge $(0 \rightarrow 2 \rightarrow 5 \rightarrow 2)$. When we update the walk counts from vertex 2, we obtain a new walk of length 3 to vertex 4 using the new edge $(0 \rightarrow 2 \rightarrow 5 \rightarrow 4)$.

Algorithm 4. Update Katz scores using dynamic algorithm given edge update *edge* from vertex *src* to *dest*

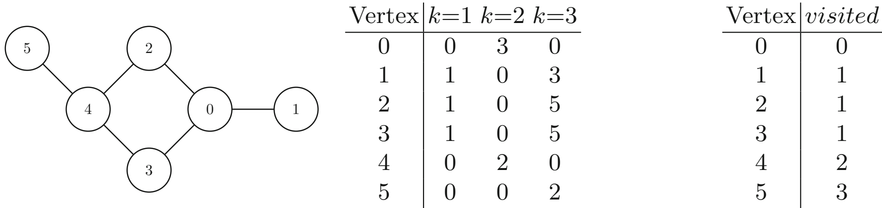
```
 1: procedure DYNAMIC_KATZ(G, seed, k, walks, visited, edge)
 2:     max_visited = MAX(visited[src],visited[dest])
 3:     if visited[src]==max_visited then
 4:         max_vtx = src; min_vtx = dest
 5:     else
 6:         max_vtx = dest; min_vtx = src
 7:     visited[max_vtx] = visited[min_vtx] + 1
 8:     walks[max_vtx][visited[max_vtx]] += 1
 9:     UPDATE_WALKS(G, max_vtx, edge, k, visited[max_vtx]+1, walks)
10:     UPDATE_WALKS(G, min_vtx, edge, k, visited[min_vtx]+1, walks)
11:     c = CALCULATE_SCORES(walks,α)
12: return c
```

Algorithm 5. Helper function for dynamic algorithm to update walks

```
 1: procedure UPDATE_WALKS(G, curr_vtx, edge, k, starting_val, walks)
 2:     map[curr_vtx] = 1
 3:     j = starting_val                ▷ Start updating walks of length starting_val
 4:     while j < k do
 5:         for v in map do
 6:             count = map[v]
 7:             for nbr in N(v) do
 8:                 if v==src AND nbr==dest then    ▷ Only update if using new edge
 9:                     map[nbr] + = count
10:         for v in map do
11:             walks[v][j] = map[v]
12:         j+ = 1
        return walks
```

3 Results

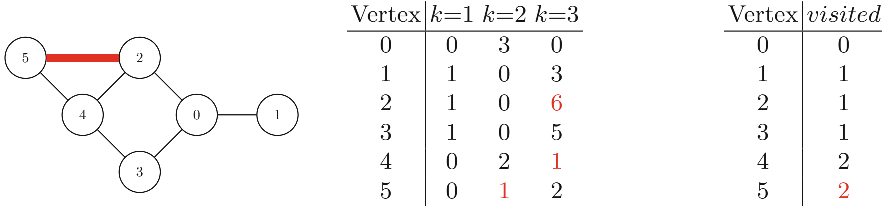
We evaluate STATIC_KATZ and DYNAMIC_KATZ on synthetic and real-world graphs. For synthetic networks, we use Erdos-Renyi graphs (ER) [16] and R-MAT graphs [17]. In the Erdos-Renyi model, all edges have the same probability for existing in the graph. R-MAT graphs are scale-free networks designed to simulate real-world graphs. For real-world networks, we use four networks from the KONECT collection [18]. Graph information is given in Table 1. For all results, five vertices from each graph are chosen randomly as seed vertices and results shown are averaged over these five seeds. Finally, many real graphs are small-world networks [19], meaning the graph diameter is on the order of $\mathcal{O}(\log(n))$, where n is again the number of vertices in the graph. Our algorithm therefore sets $k = \lceil \log(n) \rceil$, so by counting walks up to length $\approx \log(n)$, we can touch most vertices in the graph.



Vertex	k=1	k=2	k=3
0	0	3	0
1	1	0	3
2	1	0	5
3	1	0	5
4	0	2	0
5	0	0	2

Vertex	*visited*
0	0
1	1
2	1
3	1
4	2
5	3

Fig. 1. Initial graph, walk counts of length k, and visited values at time t_1.



Vertex	k=1	k=2	k=3
0	0	3	0
1	1	0	3
2	1	0	6
3	1	0	5
4	0	2	1
5	0	1	2

Vertex	*visited*
0	0
1	1
2	1
3	1
4	2
5	2

Fig. 2. Updated graph, walk counts of length k, and visited values at time t_2.

3.1 Static Results

For STATIC_KATZ, we present comparisons to the conventional linear algebraic method of computing Katz scores of solving the linear sytem $(I - \alpha A)^{-1}\mathbf{e}_i$. Recall we denote the exact solution given by linear algebra as $\mathbf{c}^*$ and $\mathbf{c}_k$ to represent the personalized Katz scores from STATIC_KATZ. Figure 3 plots the absolute error from our algorithm between $\mathbf{c}^*$ and $\mathbf{c}_k$ in the dotted blue line while the theoretically guaranteed error ϵ_k is plotted in the solid green line. Both errors are plotted as a function of k and results are averaged over all graphs. We see that the actual experimental error is always several orders of magnitude below the theoretically guaranteed error, meaning our algorithm performs better than expected.

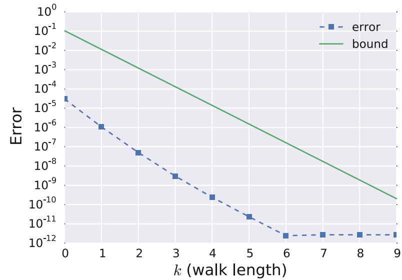


Fig. 3. Error between approximate scores $\mathbf{c}_k$ and exact solution $\mathbf{c}^*$.

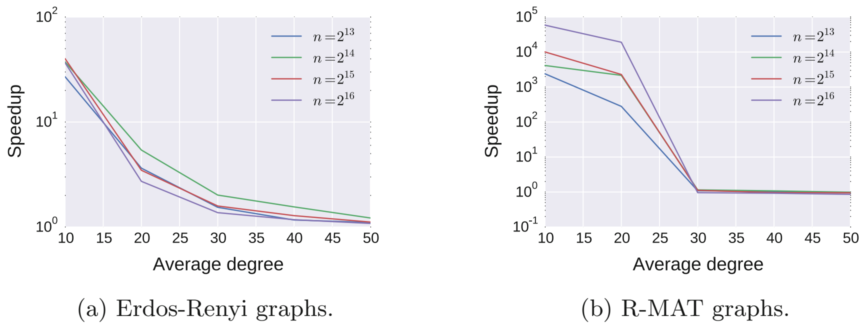
In Table 1 we summarize the relative speedup obtained from counting walks versus calculating the exact scores using linear algebra for all the real-world graphs by giving the raw times taken by both methods. Let T_L denote the time taken by the linear algebraic method and T_S the time taken by STATIC_KATZ. We note that counting walks using our method is several orders of magnitude faster than linear algebraically computing personalized Katz scores.

Table 1. Speedup for real-world networks used in experiments.

Graph	$\|V\|$	$\|E\|$	T_L	T_S
Manufacturing	167	82,927	0.74 s	0.0059 s
Facebook	42,390	876,993	132.96 s	0.0947 s
Slashdot	51,083	140,778	241.21 s	0.058 s
Digg	279,630	1,731,653	62.58 s	0.053 s

3.2 Dynamic Results

We test our method of updating Katz Centrality scores in dynamic graphs on the synthetic ER and R-MAT graphs and on the three largest real-world networks from Table 1. Dynamic results are given as comparisons to a pure static recomputation (comparing the performance and quality of DYNAMIC_KATZ to STATIC_KATZ). To have a baseline for comparison, every time we update the centrality scores using DYNAMIC_KATZ, we recompute the centrality vector statically using STATIC_KATZ. Denote the vector of scores obtained by static recomputation as $\mathbf{c}_S$ and the scores obtained by the dynamic algorithm as $\mathbf{c}_D$. 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 sequentially and apply both STATIC_KATZ and DYNAMIC_KATZ.



(a) Erdos-Renyi graphs. (b) R-MAT graphs.

Fig. 4. Speedup vs average degree for synthetic graphs tested.

For both ER and R-MAT graphs, we generate graphs with the number of vertices n as a power of 2, ranging from 2^{13} to 2^{16}. We vary the average degree of the graphs from 10 to 50. Denote the time taken by static recomputation and our dynamic algorithm as T_S and T_D respectively. We calculate speedup as T_S/T_D. Figure 4 shows the average speedup obtained over time versus the average degree in the graph. For both types of graphs we see the greatest speedup for sparser graphs (smaller average degree). For R-MAT graphs, we also observe greater speedups overall for larger graphs (larger values of n).

For real graphs, we evaluate our algorithm on the three largest graphs from Table 1. Let $S_S(R)$ and $S_D(R)$ be the sets of top R highly ranked vertices produced by static recomputation and our dynamic algorithm respectively. We evaluate the quality of our algorithm based on two metrics: (1) error $= \|\mathbf{c}_S - \mathbf{c}_D\|_2$, and (2) recall of the top R vertices $= \frac{|S_S(R) \cap S_D(R)|}{R}$. We want low values of the error, meaning DYNAMIC_KATZ produces Katz scores similar to that of STATIC_KATZ, and values of recall close to 1, meaning DYNAMIC_KATZ identifies the same highly ranked vertices as STATIC_KATZ. We consider values of R = 10, 100, and 1000. For many application purposes it is primarily the highly-ranked vertices that are of interest [20]. For example, these may be the most influential voices in a Twitter network, or sites of disease origin in a network modeling disease spread. Showing that our algorithm maintains good recall on the highly ranked vertices has many practical applications.

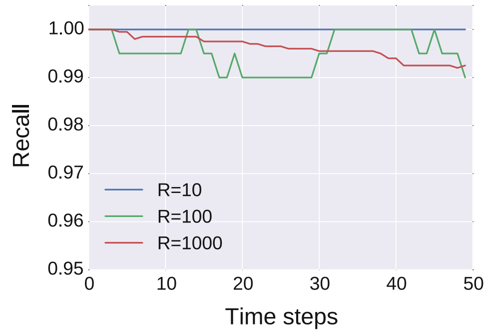


Fig. 5. Ranking accuracy over time for top R = 10,100,1000 vertices for the SLASHDOT graph.

Table 2 gives averages over time of the performance and quality of our algorithm. For the three graphs tested, our dynamic algorithm is several thousand times faster than static recomputation. Average recall of the top R vertices is very high in all cases (greater than 0.99), showing that our approximation of Katz scores is accurate enough in dynamic graphs to preserve the top highly ranked vertices in the graph. The values of the error, although relatively small, indicate that our dynamic algorithm does not find exactly the same scores as a static recomputation. Therefore, our dynamic algorithm should be used if a user's primary purpose is recall of highly ranked vertices without concern of the exact values of the scores.

Table 2. Averages over time for real-world graphs for dynamic algorithm compared to static recomputation. Columns are graph name, speedup, absolute error, and recall for R = 10, 100 and 1000.

Graph	Speedup	Average recall			Error
		$R = 10$	$R = 100$	$R = 1000$	
Facebook	27,674.50×	1.00	0.997	0.999	0.081
Slashdot	47,278.82×	1.00	0.995	0.996	0.013
Digg	60,073.81×	1.00	0.996	0.991	0.037

Furthermore, we observe that the quality of our algorithm does not suffer over time and is therefore robust to many edge insertions. Figure 5 plots the recall over time (sampled at 50 evenly spaced timepoints) for the SLASHDOT graph for the top $R = 10$, 100 and 1000 vertices. Note that the y-axis starts at 0.95. We are able to maintain a high recall of the top ranked vertices with little to no decrease over time. The results for other graphs tested are similar.

4 Conclusions

In this paper we have presented a new algorithm, STATIC_KATZ to approximate personalized Katz scores of vertices in a graph. We have shown that our approximate algorithm produces scores numerically close to, and is several orders of magnitude faster than, that of a conventional linear algebraic computation. We extended STATIC_KATZ and developed an incremental algorithm DYNAMIC_KATZ that calculated updated counts of walks to provide approximate Katz scores in dynamic graphs. Our dynamic graph algorithm is faster than a pure static recomputation and maintains high values of recall of the top ranked vertices returned. Adapting our algorithms to work in parallel is a topic for future work. For instance in our dynamic graph algorithm, updating the scores for both the source and destination vertex of the newly added edge can be done in parallel.

Acknowledgments. Eisha Nathan is in part supported by the National Physical Science Consortium Graduate Fellowship. The work depicted in this paper was sponsored in part by the National Science Foundation under award #1339745. 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

1. Caldarelli, G.: Scale-Free Networks: Complex Webs in Nature and Technology. Oxford University Press, Oxford (2007)
2. Boccaletti, S., Latora, V., Moreno, Y., Chavez, M., Hwang, D.-U.: Complex networks: structure and dynamics. Phys. Rep. **424**(4), 175–308 (2006)
3. Katz, L.: A new status index derived from sociometric analysis. Psychometrika **18**(1), 39–43 (1953)
4. Freeman, L.C.: A set of measures of centrality based on betweenness. Sociometry **40**(1), 35–41 (1977)
5. Green, O., McColl, R., Bader, D.A.: A fast algorithm for streaming betweenness centrality. In: 2012 International Conference on and 2012 International Conference on Social Computing (SocialCom), Privacy, Security, Risk and Trust (PASSAT), pp. 11–20. IEEE (2012)
6. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the web (1999)
7. Bahmani, B., Chowdhury, A., Goel, A.: Fast incremental and personalized pagerank. Proc. VLDB Endow. **4**(3), 173–184 (2010)

8. Langville, A.N., Meyer, C.D.: Updating pagerank with iterative aggregation. In: Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers and Posters, pp. 392–393. ACM (2004)
9. Riedy, J.: Updating pagerank for streaming graphs. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, pp. 877–884. IEEE (2016)
10. Foster, K.C., Muth, S.Q., Potterat, J.J., Rothenberg, R.B.: A faster katz status score algorithm. Comput. Math. Organ. Theory **7**(4), 275–285 (2001)
11. Liben-Nowell, D., Kleinberg, J.: The link-prediction problem for social networks. J. Assoc. Inf. Sci. Technol. **58**(7), 1019–1031 (2007)
12. Newman, M.: Networks: An Introduction. Oxford University Press, Oxford (2010)
13. Benzi, M., Klymko, C.: Total communicability as a centrality measure. J. Complex Netw. **1**(2), 124–149 (2013)
14. Benzi, M., Klymko, C.: A matrix analysis of different centrality measures. arXiv preprint arXiv:1312.6722 (2014)
15. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia (2003)
16. Erdös, P., Rényi, A.: On random graphs, i. Publicationes Mathematicae (Debrecen) **6**, 290–297 (1959)
17. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-mat: a recursive model for graph mining. In: SDM, vol. 4, pp. 442–446. SIAM (2004)
18. Kunegis, J.: KONECT: the koblenz network collection. In: Proceedings of the 22nd International Conference on World Wide Web, pp. 1343–1350. ACM (2013)
19. Albert, R., Jeong, H., Barabási, A.-L.: Internet: diameter of the world-wide web. Nature **401**(6749), 130–131 (1999)
20. Hawick, K., James, H.: Node importance ranking and scaling properties of some complex road networks (2007)