Scalable Katz Ranking Computation in Large Static and Dynamic Graphs

Alexander van der Grinten
Department of Computer Science, Humboldt-Universität zu Berlin, Germany
avdgrinten@hu-berlin.de

Elisabetta Bergamini
Karlsruhe Institute of Technology (KIT), Germany

Oded Green
School of Computational Science and Engineering, Georgia Institute of Technology, USA
ogreen@gatech.edu

David A. Bader
School of Computational Science and Engineering, Georgia Institute of Technology, USA
bader@gatech.edu

Henning Meyerhenke
Department of Computer Science, Humboldt-Universität zu Berlin, Germany
meyerhenke@hu-berlin.de

Abstract

Network analysis defines a number of centrality measures to identify the most central nodes in a network. Fast computation of those measures is a major challenge in algorithmic network analysis. Aside from closeness and betweenness, Katz centrality is one of the established centrality measures. In this paper, we consider the problem of computing rankings for Katz centrality. In particular, we propose upper and lower bounds on the Katz score of a given node. While previous approaches relied on numerical approximation or heuristics to compute Katz centrality rankings, we construct an algorithm that iteratively improves those upper and lower bounds until a correct Katz ranking is obtained. We extend our algorithm to dynamic graphs while maintaining its correctness guarantees. Experiments demonstrate that our static graph algorithm outperforms both numerical approaches and heuristics with speedups between 1.5× and 3.5×, depending on the desired quality guarantees. Our dynamic graph algorithm improves upon the static algorithm for update batches of less than 10000 edges. We provide efficient parallel CPU and GPU implementations of our algorithms that enable near real-time Katz centrality computation for graphs with hundreds of millions of nodes in fractions of seconds.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms, Theory of computation → Parallel algorithms

Keywords and phrases network analysis, Katz centrality, top-k ranking, dynamic graphs, parallel algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2018.42

Related Version A full version of the paper is available at [20], `https://arxiv.org/abs/1807.03847`.

Funding Most of the work was done while AvdG was affiliated with University of Cologne, Germany, and HM with Karlsruhe Institute of Technology and University of Cologne. Additionally, EB, AvdG and HM were partially supported by grant ME 3619/3-2 within German Research

26th Annual European Symposium on Algorithms (ESA 2018).
Editors: Yossi Azar, Hannah Bast, and Grzegorz Herman; Article No. 42; pp. 42:1–42:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Foundation (DFG) Priority Programme 1736. Funding was also provided by Karlsruhe House of Young Scientists via the International Collaboration Package. Funding for OG and DB was provided in part by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086. The content of the information in this document does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

1 Introduction

Finding the most important nodes of a network is a major task in network analysis. To this end, numerous centrality measures have been introduced in the literature. Examples of well-known measures are betweenness (which ranks nodes according to their participation in the shortest paths of the network) and closeness (which indicates the average shortest-path distance to other nodes). A major limitation of both measures is that they are based on the assumption that information flows through the networks following shortest paths only. However, this is often not the case in practice; think, for example, of traffic on street networks: it is easy to imagine reasons why drivers might prefer to take slightly longer paths. On the other hand, it is also quite unlikely that *much* longer paths will be taken.

Katz centrality [9] accounts for this by summing all walks starting from a node, but weighting them based on their length. More precisely, the weight of a walk of length i is α^i, where α is some attenuation factor smaller than 1. Thus, naming $\omega_i(v)$ the number of walks of length i starting from node v, the Katz centrality of v is defined as

$$\mathbf{c}(v) := \sum_{i=1}^{\infty} \omega_i(v)\, \alpha^i \tag{1}$$

or equivalently: $\mathbf{c} = \left(\sum_{i=1}^{\infty} A^i\, \alpha^i\right) \vec{I}$, where A is the adjacency matrix of the graph and $\vec{I}$ is the vector consisting only of 1s. This can be restated as a Neumann series, resulting in the closed-form expression $\mathbf{c} = \alpha A(I - \alpha A)^{-1}\vec{I}$, where I is the identity matrix. Thus, Katz centrality can be computed exactly by solving the linear system

$$(I - \alpha A)\, \mathbf{z} = \vec{I}\,, \tag{2}$$

followed by evaluating $\mathbf{c} = \alpha A\, \mathbf{z}$. We call this approach the *linear algebra formulation*. In practice, the solution to Eq. (2) is numerically approximated using iterative solvers for linear systems. While these solvers yield solutions of good quality, they can take hundreds of iterations to converge [17]. Thus, in terms of running time, those algorithms can be impractical for today's large networks, which often have millions of nodes and billions of edges.

Instead, Foster et al.'s [7] algorithm estimates Katz centrality iteratively by computing partial sums of the series from Eq. (1) until a stopping criterion is reached. Although very efficient in practice, this method has no guarantee on the correctness of the ranking it finds, not even for the top nodes. Thus, the approach is ineffective for applications where only a subset of the most central nodes is needed or when accuracy is needed. As this is indeed the case in many applications, several top-k centrality algorithms have been proposed recently for closeness [2] and betweenness [13]. Recently, a top-k algorithm for Katz centrality [17] was suggested. That algorithm still relies on solving Eq. (2); however, it reduces the numerical accuracy that is required to obtain a top-k rating. Similarly, Zhan et al. [21] propose a heuristic method to exclude certain nodes from top-k rankings but do not present algorithmic improvements on the actual Katz computation.

Dynamic graphs. Furthermore, many of today's real-world networks, such as social networks and web graphs, are dynamic in nature and some of them evolve over time at a very quick pace. For such networks, it is often impractical to recompute centralities from scratch after each graph modification. Thus, several dynamic graph algorithms that efficiently update centrality have been introduced for closeness [3] and betweenness [12]. Such algorithms usually work well in practice, because they reduce the computation to the part of the graph that has actually been affected. This offers potentially large speedups compared to recomputation. For Katz centrality, dynamic algorithms have recently been proposed by Nathan et al. [15, 16]. However, those algorithms rely on heuristics and are unable to reproduce the exact Katz ranking after dynamic updates.

Our contribution. We construct a vertex-centric algorithm that computes Katz centrality by iteratively improving upper and lower bounds on the centrality scores (see Section 3 for the construction of this algorithm). While the computed centralities are approximate, our algorithm guarantees the correct ranking. We extend (in Section 4) this algorithm to dynamic graphs while preserving the guarantees of the static algorithm. An extensive experimental evaluation (see Section 5) shows that (i) our new algorithm outperforms Katz algorithms that rely on numerical approximation with speedups between 1.5× and 3.5×, depending on the desired correctness guarantees, (ii) our algorithm has a speedup in the same order of magnitude over the widely-used heuristic of Foster et al. [7] while improving accuracy, (iii) our dynamic graph algorithm improves upon static recomputation of Katz rankings for batch sizes of less than 10000 edges and (iv) efficient parallel CPU and GPU implementations of our algorithm allow near real-time computation of Katz centrality in fractions of seconds even for very large graphs. In particular, our GPU implementation achieves speedups of more than 10× compared to a 20-core CPU implementation.

2 Preliminaries

2.1 Notation

Graphs. In the following sections, we assume that $G = (V, E)$ is the input graph to our algorithm. Unless stated otherwise, we assume that G is directed. For the purposes of Katz centrality, undirected graphs can be modeled by replacing each undirected edge with two directed edges in reverse directions. For a node $x \in V$, we denote the *out*-degree of x by $\deg(x)$. The maximum out-degree of any node in G is denoted by $\deg_{\max}$.

Katz centrality. The Katz centrality of the nodes of G is given by Eq. (1). With $\mathbf{c}_i(v)$ we denote the i-th partial sum of Eq. (1). Katz centrality is not defined for arbitrary values of α. In general, Eq. (1) converges for $\alpha < \frac{1}{\sigma_{\max}}$, where $\sigma_{\max}$ is the largest singular value of the adjacency matrix A (see [9]).

Katz centrality can also be defined by counting *inbound* walks in G [9, 18]. For this definition, $\omega_i(x)$ is replaced by the number of walks of length i that end in $x \in V$. Indeed, for applications like web graphs, nodes that are the target of many links intuitively should be considered more central than nodes that only have many links themselves[1]. However, as inbound Katz centrality coincides with the outbound Katz centrality of the reverse graph, we will not specifically consider it in this paper.

[1] This is a central idea behind the PageRank [5] metric.

2.2 Related work

Most algorithms that are able to compute Katz scores with approximation guarantees are based on the linear algebra formulation and compute a numerical solution to Eq. (2). Several approximation algorithms have been developed in order to decrease the practical running times of this formulation (e.g. based on low-rank approximation [1]). Nathan et al. [17] prove a relationship between the numerical approximation quality of Eq. (2) and the resulting Katz ranking quality. While this allows computation of top-k rankings with reduced numerical approximation quality, no significant speedups can be expected if full Katz rankings are desired.

Foster et al. [7] present a vertex-centric heuristic for Katz centrality: They propose to determine Katz centrality by computing the recurrence $c_{i+1} = \alpha A\ c_i + \vec{I}$. The computation is iterated until either a fixed point[2] or a predefined number of iterations is reached. This algorithm performs well in practice; however, due to the heuristic nature of the stopping condition, the algorithm does not give any correctness guarantees.

Another paper from Nathan et al. [16] discusses an algorithm for a "personalized" variant of Katz centrality. Our algorithm uses a similar iteration scheme but differs in multiple key properties of the algorithm: Instead of considering personalized Katz centrality, our algorithm computes the usual, "global" Katz centrality. While Nathan et al. give a global bound on the quality of their solution, we are able to compute per-node bounds that can guarantee the correctness of our ranking. Finally, Nathan et al.'s dynamic update procedure is a heuristic algorithm without correctness guarantee, although its ranking quality is good in practice. In contrast to that, our dynamic algorithm reproduces exactly the results of the static algorithm.

3 Iterative improvement of Katz bounds

3.1 Per-node bounds for Katz centrality

The idea behind our algorithm is to compute upper and lower bounds on the centrality of each node. Those bounds are iteratively improved. We stop the iteration once an application-specific stopping criterion is reached. When that happens, we say that the algorithm *converges*.

Per-node upper and lower bounds allow us to rank nodes against each other: Let $\ell_r(x)$ and $u_r(x)$ denote respectively lower and upper bounds on the Katz score of node x after iteration r. An explicit construction of those bounds will be given later in this section; for now, assume that such bounds exist. Furthermore, let w and v be two nodes; without loss of generality, we assume that w and v are chosen such that $\ell_r(w) \geq \ell_r(v)$. If $\ell_r(w) > u_r(v)$, then w appears in the Katz centrality ranking before v and we say that w and v are *separated* by the bounds ℓ_r and u_r. In this context, it should be noted that per-node bounds do not allow us to prove that the Katz scores of two nodes are equal[3]. However, as the algorithm still needs to be able to rank nodes x that share the same $\ell_r(x)$ and $u_r(x)$ values, we need a more relaxed concept of separation. Therefore:

▶ **Definition 1.** In the same setting as before, let $\epsilon > 0$. We say that w and v are ϵ-*separated*, if and only if

$$\ell_r(w) > u_r(v) - \epsilon\ . \tag{3}$$

[2] Note that a true fixed point will not be reached using this method unless the graph is a DAG.

[3] In theory, the linear algebra formulation is able to prove that the score of two nodes is indeed equal. However, in practice, limited floating point precision limits the usefulness of this property.

Intuitively, the introduction of ϵ makes the ϵ-condition easier to fulfill than the separation condition: Indeed, separated pairs of nodes are also ϵ-separated for every $\epsilon > 0$. In particular, ϵ-separation allows us to construct Katz rankings even in the presence of nodes that have the same Katz score: Those nodes are never separated, but they will eventually be ϵ-separated for every $\epsilon > 0$.

In order to actually construct rankings, it is sufficient to notice that once all pairs of nodes are ϵ-separated, sorting the nodes by their lower bounds ℓ_r yields a correct Katz ranking, except for pairs of nodes with a difference in Katz score of less than ϵ. Thus, using this definition, we can discuss possible stopping criteria for the algorithm:

Ranking criterion. Stop once all nodes are ϵ-separated from each other. This guarantees that the ranking is correct, except for nodes with scores that are very close to each other.

Top-k criterion. Stop once the top-k nodes are ϵ-separated from each other and from all other nodes. For $k = n$ this criterion reduces to the ranking criterion.

Score criterion. Stop once the difference between the upper and lower bound of each node becomes less than ϵ. This guarantees that the Katz centrality of each node is correct up to an additive constant of ϵ.

Pair criterion. Stop once two given nodes u and v are ϵ-separated.

First, we notice that a simple lower bound on the Katz centrality of a node v can be obtained by truncating the series in Eq. (1) after r iterations, hence, $\ell_r(v) := \sum_{i=1}^{r} \omega_i(v)\alpha^i$ is a lower bound on $\mathbf{c}(v)$. For undirected graphs, this lower bound can be improved to $\sum_{i=1}^{r} \omega_i(v)\alpha^i + \omega_r(v)\alpha^{r+1}$, as any walk of length r can be extended to a walk of length $r+1$ with the same starting point by repeating its last edge with reversed direction.

▶ **Theorem 2.** *Let $\gamma = \frac{\deg_{\max}}{1-\alpha\deg_{\max}}$. For any $r \geq 1$, $v \in V$ and $\alpha < \frac{1}{\deg_{\max}}$, the value*

$$u_r(v) := \sum_{i=1}^{r} \alpha^i \omega_i(v) + \alpha^{r+1}\omega_r(v)\gamma$$

is an upper bound on $\mathbf{c}(v)$.

Proof. First, let $S_i(v)$ be the set of nodes x for which there exists a walk of length i starting in v and ending in x. Each walk of length $i+1$ is the concatenation of a walk of length i ending in $x \in S_i(v)$ and an edge (x, y), where y is some neighbor of x. Let $\omega_i(v, x)$ denote the number of walks of length i that start in v and end in x. Thus, we can write

$$\omega_{i+1}(v) = \sum_{x \in S_i(v)} \deg(x)\ \omega_i(v, x) \leq \sum_{x \in S_i(v)} \deg_{\max}\ \omega_i(v, x) = \deg_{\max}\ \omega_i(v)\ . \tag{4}$$

By applying induction to the previous inequality, it is easy to see that, for any $j > 1$,

$$\omega_{i+j}(v) \leq (\deg_{\max})^j \omega_i(v)\ .$$

Discarding the first r terms of the sum in Eq. (1) then yields

$$\begin{aligned}\sum_{i=r+1}^{\infty} \alpha^i \omega_i(v) &\leq \sum_{j=1}^{\infty} \alpha^{r+j} (\deg_{\max})^j \omega_r(v) = \alpha^r \omega_r(v) \sum_{j=1}^{\infty} (\alpha \deg_{\max})^j \\ &= \alpha^r \omega_r(v) \left(\frac{1}{1 - \alpha\deg_{\max}} - 1 \right) = \alpha^{r+1} \omega_r(v)\gamma\ .\end{aligned}$$

For the second to last equality, we rewrite the infinite series as a geometric sum. ◀

The following lemma (proof in the full version of this paper [20]) shows that we can indeed iteratively improve the upper and lower bounds for each node $x \in V$:

▶ **Lemma 3.** *For each $x \in V$, $\ell_i(x)$ is non-decreasing in i and $u_i(x)$ is non-increasing in i.*

Theorem 2 requires us to choose $\alpha < \frac{1}{\deg_{\max}}$, which is a restriction compared to the more general requirement of $\alpha < \frac{1}{\sigma_{\max}}$. For our experiments in the later sections of this paper, we set $\alpha = \frac{1}{1+\deg_{\max}}$ in order to satisfy this condition. Aside from enabling us to apply the theorem, this choice of α has some additional advantages: First, because Theorem 2 gives an upper bound on Eq. (1), Katz centrality is guaranteed to converge for this value of the α parameter[4]. $\deg_{\max}$ is also much easier to compute than $\sigma_{\max}$, an operation that is comparable in complexity to computing the Katz centrality itself[5]. Finally, $\alpha = \frac{1}{1+\deg_{\max}}$ is widely-used in existing literature [4, 7], with Foster et al. calling it the "generally-accepted default attenuation factor" [7].

It is worth remarking (proof in the full version of this paper [20]) that graphs exist for which the bound from Theorem 2 is sharp:

▶ **Lemma 4.** *If G is a complete graph, $u_i(x) = \mathbf{c}(x)$ for all $x \in V$ and $i \in \mathbb{N}$.*

3.2 Efficient rankings using per-node bounds

In the following, we state the description of our Katz algorithm for static graphs. As hinted earlier, the algorithm estimates Katz centrality by computing $u_r(v)$ and $\ell_r(v)$. These upper and lower bounds are iteratively improved by incrementing r until the algorithm converges.

To actually compute $\mathbf{c}_r(v)$, we use the well-known fact that the number of walks of length i starting in node v is equal to the sum of the number of walks of length $i-1$ starting in the neighbors of v, in other words:

$$\omega_i(v) = \sum_{v \to x \in E} \omega_{i-1}(x) \ . \tag{5}$$

Thus, if we initialize $\omega_1(v)$ to $\deg(v)$ for all $v \in V$, we can then repeatedly loop over the edges of G and compute tighter and tighter lower bounds.

We focus here on the top-k convergence criterion. It is not hard to see how our techniques can be adopted to the other stopping criteria mentioned at the start of the previous subsection. To be able to efficiently detect convergence, the algorithm maintains a set of *active* nodes. These are the nodes for which the lower and upper bounds have not yet converged. Initially, all nodes are active. Each node is *deactivated* once it is ϵ-separated from the k nodes with highest lower bounds ℓ_r. It should be noted that, because of Lemma 3, deactivated nodes will stay deactivated in all future iterations. Thus, for the top-k criterion, it is sufficient to check whether (i) only k nodes remain active and (ii) the remaining active nodes are ϵ-separated from each other. This means that each iteration will require less work than its previous iteration.

Algorithm 1 depicts the pseudocode of the algorithm. Computation of $\omega_r(v)$ is done by evaluating the recurrence from Eq. (5). After the algorithm terminates, the ϵ-separation property guarantees that the k nodes with highest $\ell_r(v)$ form a top-k Katz centrality ranking (although $\ell_r(v)$ does not necessarily equal the true Katz score).

[4] This was already noticed by Katz [9] and can alternatively be proven through linear algebra.

[5] Indeed, the popular power iteration method to compute $\sigma_{\max}$ for real, symmetric, positive-definite matrices has a complexity of $\Omega(r\,|E|)$, where r denotes a number of iterations.

Algorithm 1 Katz centrality bound computation for static graphs.

$\gamma \leftarrow \deg_{\max} / (1 - \alpha \deg_{\max})$
Initialize $\mathbf{c}_0(x) \leftarrow 0 \quad \forall x \in V$
Initialize $r \leftarrow 0$ and $\omega_0(x) \leftarrow 1 \quad \forall x \in V$
Initialize set of active nodes: $M \leftarrow V$
while not CONVERGED() **do**
  Set $r \leftarrow r + 1$ and $\omega_r(x) \leftarrow 0 \quad \forall x \in V$
  for all $v \in V$ **do**
    for all $v \to u \in E$ **do**
      $\omega_r(v) \leftarrow \omega_r(v) + \omega_{r-1}(u)$
    $\mathbf{c}_r(v) \leftarrow \mathbf{c}_{r-1}(v) + \alpha^r \omega_r(v)$
    if G undirected **then**
      $\ell_r(v) \leftarrow \mathbf{c}_r(v) + \alpha^{r+1} \omega_r(v)$
    else
      $\ell_r(v) \leftarrow \mathbf{c}_r(v)$
    $u_r(v) \leftarrow \mathbf{c}_r(v) + \alpha^{r+1} \omega_r(v) \gamma$

function CONVERGED()
  PARTIALSORT(M, k, ℓ_r, decreasing)
  for all $i \in \{k+1, \ldots, |V|\}$ **do**
    if $u_r(M[i]) - \epsilon < \ell_r(M[k])$ **then**
      $M \leftarrow M \setminus \{v\}$
  if $|M| > k$ **then**
    return false
  for all $i \in \{2, \ldots, \min(|M|, k)\}$ **do**
    if $u_r(M[i]) - \epsilon \geq \ell_r(M[i-1])$ **then**
      return false
  return true

The CONVERGED procedure in Algorithm 1 checks whether the top-k convergence criterion is satisfied. In this procedure, M denotes the set of active nodes. The procedure first partially sorts the elements of M by decreasing lower bound ℓ_r. After that is done, the first k elements of M correspond to the top-k elements in the current ranking (which might not be correct yet). Note that it is not necessary to construct the entire ranking here; sorting just the top-k nodes is sufficient. The procedure tries to deactivate nodes that cannot be in the top-k and afterwards checks if the remaining top-k nodes are correctly ordered. These checks are performed by testing if the ϵ-separation condition from Eq. (3) is true.

Complexity analysis. The sequential worst-case time complexity of Algorithm 1 is $\mathcal{O}(r\,|E| + r\,\mathcal{C})$, where r is the number of iterations and $\mathcal{C}$ is the complexity of the convergence checking procedure. It is easy to see that the loop over V can be parallelized, yielding a complexity of $\mathcal{O}(r\,\frac{|V|}{p}\,\deg_{max} + r\,\mathcal{C})$ on a parallel machine with p processors. The complexity of CONVERGED, the top-k ranking convergence criterion, is dominated by the $\mathcal{O}(|V| + k \log k)$ complexity of partial sorting. Both the score and the pair criteria can be implemented in $\mathcal{O}(1)$.

It should be noted that – for the same solution quality – our algorithm converges at least as fast as the heuristic of Foster et al. that computes a Katz ranking without correctness guarantee. Indeed, the values of $\mathbf{c}_r$ yield exactly the values that are computed by the heuristic. However, Foster et al.'s heuristic is unable to accurately assess the quality of its current solution and might thus perform too many or too few iterations.

4 Updating Katz centrality in dynamic graphs

In this section, we discuss how our Katz centrality algorithm can be extended to compute Katz centrality rankings for dynamically changing graphs. We model those graphs as an initial graph that is modified by a sequence of edge insertions and edge deletions. We do not explicitly handle node insertions and deletions as those can easily be supported by adding enough isolated nodes to the initial graph.

Before processing any edge updates, we assume that our algorithm from Section 3 was first executed on the initial graph to initialize the values $\omega_i(x)$ for all $x \in V$. The dynamic graph algorithm needs to recompute $\omega_i(x)$ for $i \in \{1, \ldots, r\}$, where r is the number of iterations that was reached by the static Katz algorithm on the initial graph. The main observation here is that if an edge $u \to v$ is inserted into (or deleted from) the initial graph, $\omega_i(x)$ only changes for nodes x in the vicinity of u. More precisely, $\omega_i(x)$ can only change if u is reachable from x in at most $i - 1$ steps.

Algorithm 2 Dynamic Katz update procedure.

```
E ← E \ D
S ← ∅, T ← ∅
for all w → v ∈ I ∪ D do
    S ← S ∪ {w}
    T ← T ∪ {v}
for all i ∈ {1, ..., r} do
    updateLevel(i)
for all w ∈ S do
    Recompute ℓ_r(w) and u_r(w) from c_r(w)
for all w ∈ V do
    if u_r(w) ≥ min_{x∈M} ℓ_r(x) − ε then
        M ← M ∪ {w}                    ▷ Reactivation
E ← E ∪ I
while not converged() do
    Run more iterations of static algorithm

procedure updateLevel(i)
    for all v ∈ S ∪ T do
        ω'_i(v) ← ω_i(v)
    for all v ∈ S do
        for all w → v ∈ E do
            S ← S ∪ {w}
            ω'_i(w) ← ω'_i(w) − ω_{i−1}(v) + ω'_{i−1}(v)
    for all w → v ∈ I do
        ω'_i(w) ← ω'_i(w) + ω'_{i−1}(v)
    for all w → v ∈ D do
        ω'_i(w) ← ω'_i(w) − ω_{i−1}(v)
    for all w ∈ S do
        c_i(w) ← c_i(w) − α^i ω_i(w) + α^i ω'_i(w)
```

Algorithm 2 depicts the pseudocode of our dynamic Katz algorithm. $\mathcal{I}$ denotes the set of edges to be inserted, while $\mathcal{D}$ denotes the set of edges to be deleted. We assume that $\mathcal{I} \cap E = \emptyset$ and $\mathcal{D} \subseteq E$ before the algorithm. Effectively, the algorithm performs a breadth-first search (BFS) through the reverse graph of G and updates ω_i for all nodes nodes that were reached in steps 1 to i.

After the update procedure terminates, the new upper and lower bounds can be computed from $\mathbf{c}_r$ as in the static algorithm. We note that $\omega'_i(x)$ matches exactly the value of $\omega_i(x)$ that the static Katz algorithm would compute for the modified graph. Hence, the dynamic algorithm reproduces the correct values of $\mathbf{c}_r(x)$ and also of $\ell_r(x)$ and $u_r(x)$ for all $x \in V$. In case of the top-k convergence criterion, some nodes might need to be *reactivated* afterwards: Remember that the top-k criterion maintains a set M of active nodes. After edge updates are processed, it can happen that there are nodes x that are not ϵ-separated from all nodes in M anymore. Such nodes x need to be added to M in order to obtain a correct ranking. The ranking itself can then be updated by sorting M according to decreasing ℓ_r.

It should be noted that there is another related corner case: Depending on the convergence criterion, it can happen that the algorithm is not converged anymore even after nodes have been reactivated. For example, for the top-k criterion, this is the case if the nodes in M are not ϵ-separated from each other anymore. Thus, after the dynamic update we have to perform a convergence check and potentially run additional iterations of the static algorithm until it converges again.

Assuming that no further iterations of the static algorithms are necessary, the complexity of the update procedure is $\mathcal{O}(r\,|E| + \mathcal{C})$, where $\mathcal{C}$ is the complexity of convergence checking (see Section 3). In reality, however, the procedure can be expected to perform much better: Especially for the first few iterations, we expect the set S of vertices visited by the BFS to be much smaller than $|V|$. However, this implies that effective parallelization of the dynamic graph algorithm is more challenging than the static counterpart. We mitigate this problem, by aborting the BFS if $|S|$ becomes large and just update the ω_i scores unconditionally for all nodes.

Finally, it is easy to see that the algorithm can be modified to update ω in-place instead of constructing a new ω' matrix. For this optimization, the algorithm needs to save the value of ω_i for all nodes of S before overwriting it, as this value is required for iteration $i + 1$. For readability, we omit this modification in the pseudocode.

Table 1 Performance of the Katz algorithm, ranking criterion.

ϵ	r[a)]	Runtime[a)]	Separation[b)]	ϵ	r[a)]	Runtime[a)]	Separation[b)]
10^{-1}	2.3	33.51 s	96.189974 %	10^{-7}	7.2	78.74 s	99.994959 %
10^{-2}	3.0	42.81 s	98.478250 %	10^{-8}	7.9	83.28 s	99.998866 %
10^{-3}	3.8	51.59 s	99.264726 %	10^{-9}	8.6	85.10 s	99.998886 %
10^{-4}	4.8	65.99 s	99.391884 %	10^{-10}	9.2	89.03 s	99.998889 %
10^{-5}	5.7	71.53 s	99.992908 %	10^{-11}	9.8	99.43 s	99.998934 %
10^{-6}	6.5	70.59 s	99.994861 %	10^{-12}	10.4	96.86 s	99.998934 %
Foster	11.2	105.03 s	-	CG	12.0	117.24 s	-

a) Average over all instances. r is the number of iterations.
b) Fraction of node pairs that are separated (and not only ϵ-separated). Lower bound on the correctly ranked pairs. This is the geometric mean over all graphs.

5 Experiments

Implementation details. The new algorithm in this paper is hardware independent and as such we can implement it on different types of hardware with the right type of software support. Specifically, our dynamic Katz centrality requires a dynamic graph data structure. On the CPU we use NetworKit [19]; on the GPU we use Hornet[6]. The Hornet data structure is architecture independent, though at time of writing only a GPU implementation exists.

NetworKit consists of an optimized C++ network analysis library and bindings to access this library from Python. NetworKit contains parallel shared-memory implementations of many popular graph algorithms and can handle networks with billions of edges.

The Hornet [6], an efficient extension to the cuSTINGER [8] data structure, is a dynamic graph and matrix data structure designed for large scale networks and to support graphs with trillions of vertices. In contrast to cuSTINGER, Hornet better utilizes memory, supports memory reclamation, and can be updated almost ten times faster.

In our experiments, we compare our new algorithm to Foster et al.'s heuristic and a conjugate gradient (CG) algorithm (without preconditioning) that solves Eq. (2). The performance of CG could be possibly improved by employing a suitable preconditioner; however, we do not expect this to change our results qualitatively. Both of these algorithms were implemented in NetworKit and share the graph data structure with our new Katz implementation. We remark that for the static case, both CG and our Katz algorithm could be implemented on top of a CSR matrix data structure to improve the data locality and speed up the implementation.

Experimental setup. We evaluate our algorithms on a set of complex networks. The networks originate from diverse real-world applications and were taken from SNAP [14] and KONECT [11]. Details about the exact instances that we used can be found in the full version of this paper [20]. In order to be able to compare our algorithm to the CG algorithm, we turn the directed graphs in this test set into undirected graphs by ignoring edge directions. This ensures that the adjacency matrix is symmetric and CG is applicable. Our new algorithm itself would be able to handle directed graphs just fine.

[6] Hornet can be found at `https://github.com/hornet-gt`, while NetworKit is available from `https://github.com/kit-parco/networkit`. Both projects are open source, including the implementations of our new algorithm.

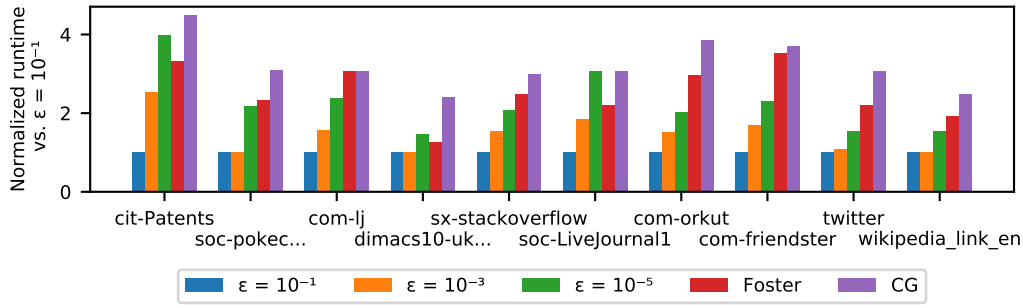

Figure 1 Katz performance on individual instances.

All CPU experiments ran on a machine with dual-socket Intel Xeon E5-2690 v2 CPUs with 10 cores per socket[7] and 128 GiB RAM. Our GPU experiments are conducted on an NVIDIA P100 GPU which has 56 Streaming Multiprocessors (SMs) and 64 Streaming Processors (SPs) per SM (for a total of 3584 SPs) and has 16GB of HBM2 memory. To effectively use the GPU, the number of active threads need to be roughly 8 times larger than the number of SPs. The Hornet framework has an API that enables such parallelization (with load balancing) such that the user only needs to write a few lines of code.

5.1 Evaluation of the static Katz algorithm

In a first experiment, we evaluate the running time of our static Katz algorithm. In particular, we compare it to the running time of the linear algebra formulation (i.e. the CG algorithm) and Foster et al.'s heuristic. We run CG until the residual is less than 10^{-15} to obtain a nearly exact Katz ranking (i.e. up to machine precision; later in this section, we compare to CG runs with larger error tolerances). For Foster's heuristic, we use an error tolerance of 10^{-9}, which also yields an almost exact ranking. For our own algorithm, we use the ranking convergence criterion (see Section 3) and report running times and the quality of our correctness guarantees for different values of ϵ. All algorithms in this experiment ran in single-threaded mode.

Table 1 summarizes the results of the evaluation. The fourth column of Table 1 states the fraction of separated pairs of nodes. This value represents a lower bound on the correctness of ranking. Note that pairs of nodes that have the same Katz score will never be separated. Indeed, this seems to be the case for about 0.001% of all pairs of nodes (as they are never separated, not even if ϵ is very low). Taking this into account, we can see that our algorithm already computes the correct ranking for 99% of all pairs of nodes at $\epsilon = 10^{-3}$. At this ϵ, our algorithm outperforms the other Katz algorithms considerably.

Furthermore, Table 1 shows that the average running time of our algorithm is smaller than the running time of the Foster et al. and CG algorithms. However, the graphs in our instance set vastly differ in size and originate from different applications; thus, the average running time alone does not give good indication for performance on individual graphs. In Figure 1 we report running times of our algorithm for the ten largest individual instances. $\epsilon = 10^{-1}$ is taken as baseline and the running times of all other algorithms are reported relative to this baseline. In the $\epsilon \leq 10^{-3}$ setups, our Katz algorithm outperforms the CG and

[7] Hyperthreading was disabled for the experiments.

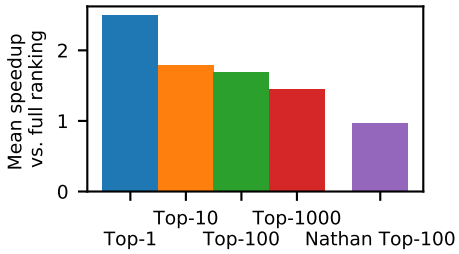

Figure 2 Top-k speedup over full ranking.

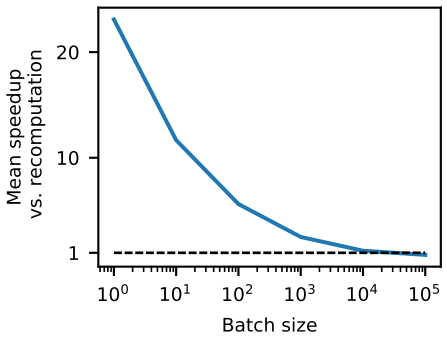

Figure 3 Dynamic update performance.

Foster et al. algorithms on all instances. Foster et al.'s algorithm is faster than our algorithm for $\epsilon = 10^{-5}$ on three out of ten instances. On the depicted instances, CG is never faster than our algorithm, although it can outperform our algorithm on some small instances and for very low ϵ.

Finally, in Figure 2, we present results of our Katz algorithm while using the top-k convergence criterion. We report (geometric) mean speedups relative to the full ranking criterion. The figure also includes the approach of Nathan et al. [17]. Nathan et al. conducted experiments on real-world graphs and concluded that solving Eq. (2) with an error tolerance of 10^{-4} in practice almost always results in the correct top-100 ranking. Thus, we run CG with that error tolerance. However, it turns out that this approach is barely faster than our full ranking algorithm. In contrast to that, our top-k algorithm yields decent speedups for $k \leq 1000$.

5.2 Evaluation of the dynamic Katz algorithm

In our next experiment, we evaluate the performance of our dynamic Katz algorithm to compute top-1000 rankings using $\epsilon = 10^{-4}$. We select b random edges from the graph, delete them in a single batch and run our dynamic update algorithm on the resulting graph. We vary the batch size b from 10^0 to 10^5 and report the running times of the dynamic graph algorithm relative to recomputation. Similar to the previous experiment, we run the algorithms in single-threaded mode. Note that while we only show results for edge deletion, edge insertion is completely symmetric in Algorithm 2.

Figure 3 summarizes the results of the experiment. For batch sizes $b \leq 1000$, our dynamic algorithm offers a considerable speedup over recomputation of Katz centralities. As many of the graphs in our set of instances have a small diameter, for larger batch sizes ($b > 10000$), almost all of the vertices of the graph need to be visited during the dynamic update procedure. Hence, the dynamic update algorithm is slower than recomputation in these cases.

5.3 Real-time Katz computation using parallel CPU and GPU implementations

Our last experiment concerns the practical running time and scalability of efficient parallel CPU and GPU implementations of our algorithm. For this, we compare the running times of our shared-memory CPU implementation with different numbers of cores. Furthermore, we report results of our GPU implementation. Because of GPU memory constraints, we could

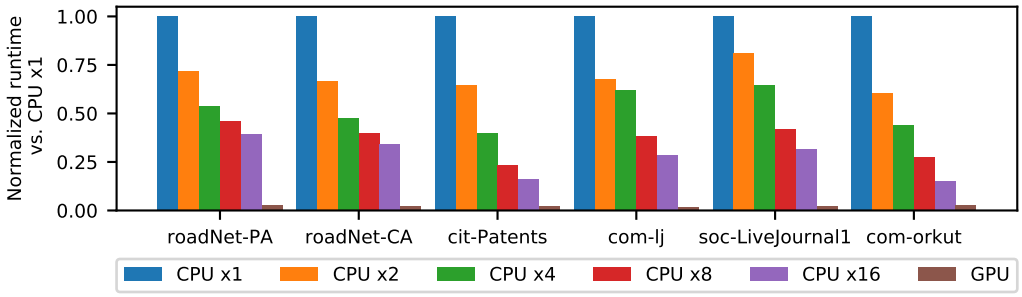

Figure 4 Scalability of parallel CPU and GPU implementations.

not process all of the graphs on the GPU. Hence, we provide the results of this experiment only for a subset of graphs that do fit into the memory of our GPU. The graphs in this subset have between 1.5 million and 120 million edges. We use the top-10000 convergence criterion with $\epsilon = 10^{-6}$.

Figure 4 depicts the results of the evaluation. In this figure, we consider the sequential CPU implementation as a baseline. We report the relative running times of the 2, 4, 8 and 16 core CPU configurations, as well as the GPU configuration, to this baseline. While the parallel CPU configurations yield moderate speedups over the sequential implementation, the GPU gives a significant speedup over the 16 core CPU configuration[8]. Even compared to a 20 core CPU configuration (not depicted in the plots; see the full version of this paper [20]), the GPU achieves a (geometric) mean speedup of 10×.

The CPU implementation achieves running times in the range of seconds; however, our GPU implementation reduces this running time to a fraction of a second. In particular, the GPU running time varies between 20 ms (for roadNet-PA) and 213 ms (for com-orkut), enabling near real-time computation of Katz centrality even for graphs with hundreds of millions of edges.

6 Conclusion

In this paper, we have presented an algorithm for Katz centrality that computes upper and lower bounds on the Katz score of individual nodes. Experiments demonstrated that our algorithm outperforms both linear algebra formulations and approximation algorithms, with speedups between 150% and 350% depending on desired correctness guarantees.

Future work could try to provide stricter per-node bounds for Katz centrality to further decrease the number of iterations that the algorithm requires to convergence. In particular, it would be desirable to prove per-node bounds that do not rely on $\alpha < 1/\deg_{\max}$. On the implementation side, our new algorithm could be formulated in the language of GraphBLAS [10] to enable it to run on a variety of upcoming software and hardware architectures.

[8] At time of writing, our CPU implementation uses a sequential algorithm for partial sorting; this is a bottleneck in the parallel CPU configurations.

References

1 E. Acar, D. M. Dunlavy, and T. G. Kolda. Link prediction on evolving data using matrix and tensor factorizations. In *2009 IEEE International Conference on Data Mining Workshops*, pages 262–269, Dec 2009. doi:10.1109/ICDMW.2009.54.

2 Elisabetta Bergamini, Michele Borassi, Pierluigi Crescenzi, Andrea Marino, and Henning Meyerhenke. Computing top-k closeness centrality faster in unweighted graphs. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 68–80. Society for Industrial and Applied Mathematics, 2018. doi:10.1137/1.9781611974317.6.

3 Patrick Bisenius, Elisabetta Bergamin, Eugenio Angriman, and Henning Meyerhenke. Computing top-k closeness centrality in fully-dynamic graphs. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 21–35. Society for Industrial and Applied Mathematics, 2018. doi:10.1137/1.9781611975055.3.

4 Francesco Bonchi, Pooya Esfandiar, David Gleich, Chen Greif, and Laks Lakshmanan. Fast matrix computations for pairwise and columnwise commute times and katz scores. *Internet Mathematics*, 8:73–112, 03 2012.

5 Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117, 1998. Proceedings of the Seventh International World Wide Web Conference. doi:10.1016/S0169-7552(98)00110-X.

6 F. Busato, O. Green, N. Bombieri, and D.A. Bader. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *IEEE Proc. High Performance Extreme Computing (HPEC)*, Waltham, MA, 2018.

7 Kurt C. Foster, Stephen Q. Muth, John J. Potterat, and Richard B. Rothenberg. A faster katz status score algorithm. *Computational & Mathematical Organization Theory*, 7(4):275–285, Dec 2001. doi:10.1023/A:1013470632383.

8 O. Green and D.A. Bader. cuSTINGER: Supporting Dynamic Graph Algorithms for GPUs. In *IEEE Proc. High Performance Embedded Computing Workshop (HPEC)*, Waltham, MA, 2016.

9 Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, Mar 1953. doi:10.1007/BF02289026.

10 J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira. Mathematical foundations of the graphblas. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, Sept 2016. doi:10.1109/HPEC.2016.7761646.

11 Jérôme Kunegis. Konect: The koblenz network collection. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13 Companion, pages 1343–1350, New York, NY, USA, 2013. ACM. doi:10.1145/2487788.2488173.

12 Min-Joong Lee, Sunghee Choi, and Chin-Wan Chung. Efficient algorithms for updating betweenness centrality in fully dynamic graphs. *Information Sciences*, 326:278–296, 2016. doi:10.1016/j.ins.2015.07.053.

13 Min-Joong Lee and Chin-Wan Chung. Finding k-highest betweenness centrality vertices in graphs. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, pages 339–340, New York, NY, USA, 2014. ACM. doi:10.1145/2567948.2577358.

14 Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

15 Eisha Nathan and David A. Bader. 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 2017, ASONAM '17, pages 149–154, New York, NY, USA, 2017. ACM. doi:10.1145/3110025.3110034.

16 Eisha Nathan and David A. Bader. Approximating personalized katz centrality in dynamic graphs. In Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 290–302, Cham, 2018. Springer International Publishing.

17 Eisha Nathan, Geoffrey Sanders, James Fairbanks, Van Emden Henson, and David A. Bader. Graph ranking guarantees for numerical approximations to katz centrality. *Procedia Computer Science*, 108:68–78, 2017. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland. doi:10.1016/j.procs.2017.05.021.

18 Mark Newman. *Networks: An Introduction*. Oxford University Press, Inc., New York, NY, USA, 2010.

19 Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. NetworKit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016. doi:10.1017/nws.2016.20.

20 A. van der Grinten, E. Bergamini, O. Green, D.A. Bader, and Henning Meyerhenke. Scalable Katz ranking computation in large static and dynamic graphs. *arXiv*, 2018. arXiv:1807.03847.

21 Justin Zhan, Sweta Gurung, and Sai Phani Krishna Parsa. Identification of top-k nodes in large networks using katz centrality. *Journal of Big Data*, 4(1):16, May 2017. doi:10.1186/s40537-017-0076-5.