Scalable Katz Ranking Computation in Large Static and Dynamic Graphs

ALEXANDER VAN DER GRINTEN, Humboldt-Universität zu Berlin
ELISABETTA BERGAMINI, Karlsruhe Institute of Technology
ODED GREEN, NVIDIA
DAVID A. BADER, New Jersey Institute of Technology
HENNING MEYERHENKE, Humboldt-Universität zu Berlin

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 article, 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. Previous approaches relied on numerical approximation or heuristics to compute Katz centrality rankings; however, 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 10,000 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 edges in fractions of seconds.

CCS Concepts: • **Theory of computation → Dynamic graph algorithms**; *Parallel algorithms;*

Additional Key Words and Phrases: Network analysis, Katz centrality, top-k ranking, dynamic graphs, parallel algorithms

A shorter version of this article was published at the European Symposium on Algorithms (ESA) 2018 [22].
This work was done while A. van der Grinten and H. Meyerhenke were affiliated with the University of Cologne. E. Bergamini, A. van der Grinten, and H. Meyerhenke were partially supported by grant ME 36119/3-2 within German Research Foundation (DFG) Priority Programme 1736 and DFG grant ME 3619/4-1. A. van der Grinten was partially supported by DFG grant GR 5745/1-1. Funding was also provided by Karlsruhe House of Young Scientists via the International Collaboration Package. Funding for O. Green and D. A. Bader 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.
Authors' addresses: A. van der Grinten and H. Meyerhenke, Department of Computer Science, Humboldt-Universität zu Berlin, Germany; emails: {avdgrinten, meyerhenke}@hu-berlin.de; E. Bergamini, Karlsruhe Institute of Technology, Germany; email: elisabetta.bergamini89@gmail.com; O. Green, NVIDIA; email: ogreen@nvidia.com; D. A. Bader, Institute for Data Science, New Jersey Institute of Technology; email: bader@njit.edu.

1084-6654/2022/07-ART1.7
https://doi.org/10.1145/3524615

ACM Reference format:
Alexander van der Grinten, Elisabetta Bergamini, Oded Green, David A. Bader, and Henning Meyerhenke. 2022. Scalable Katz Ranking Computation in Large Static and Dynamic Graphs. *J. Exp. Algorithmics* 27, 1, Article 1.7 (July 2022), 16 pages.
https://doi.org/10.1145/3524615

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. These centrality measures usually compute a numerical score for each vertex of the graph. The resulting scores are then used to obtain a ranking of the vertices. In many applications, domain scientists are interested in the centrality ranking only (and not in the actual centrality scores) [20].

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 the reasons drivers might prefer to take slightly longer paths. Yet it is also quite unlikely that much longer paths will be taken.

Katz centrality [11] 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: $\mathbf{c} = (\sum_{i=1}^{\infty} A^i\, \alpha^i)\vec{I}$, 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 Equation (2) is numerically approximated using iterative solvers for linear systems. Although these solvers yield solutions of good quality, they can take hundreds of iterations to converge [19]. 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, the algorithm of Foster et al. [9] estimates Katz centrality iteratively by computing partial sums of the series from Equation (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 [4] and betweenness [15]. Recently, a top-k algorithm for Katz centrality [19] was suggested. That algorithm still relies on solving Equation (2); however, it reduces the numerical accuracy that is required to obtain a top-k rating. Similarly, Zhan et al. [23] 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 centrality measures from scratch after each graph modification. Thus, several dynamic graph algorithms that efficiently update centrality have been introduced for closeness [5] and betweenness [14]. 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 and Bader [17, 18]. 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). Although the computed centrality scores 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. [9] while improving accuracy; (iii) our dynamic graph algorithm improves upon static recomputation of Katz rankings for batch sizes of less than 10,000 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.

The variants of the algorithm that we describe in this article are able to handle general attenuation factors α (i.e., values of α up to the largest valid value of $\frac{1}{\sigma_{\max}}$), whereas the variant that was previously published in our conference paper [22] can only handle $\alpha < \frac{1}{\deg_{\max}}$. This is achieved via a new *spectral* bound (in Section 3.2) on the Katz centrality score. Experiments (in Section 5.2) confirm that our algorithm is also faster than numerical solvers when the spectral bound is used.

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 opposite 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 Equation (1). With $\mathbf{c}_i(v)$, we denote the i-th partial sum of Equation (1). Katz centrality is not defined for arbitrary values of α. In general, Equation (1) converges for $\alpha < \frac{1}{\sigma_{\max}}$, where $\sigma_{\max}$ is the largest singular value of the adjacency matrix A (see the work of Katz [11]). Katz centrality can also be defined by counting *inbound* walks in G [11, 20]. 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

[1] This is a central idea behind the PageRank [6] metric.

outbound Katz centrality of the reverse graph, we will not specifically consider it in this article.

Matrix norms. Let M be an $\mathbb{R}$-valued $n \times n$ matrix. Let $||M||_p$ denote the matrix norm induced by the ℓ_p-norm on $\mathbb{R}^n$: $||M||_p = \sup \{\frac{||Mx||_p}{||x||_p} : x \in \mathbb{R}^n, x \neq 0\}$. It holds that $||MM'||_p \leq ||M||_p||M'||_p$ and hence $||M^i||_p \leq ||M||_p^i$ for $i \in \mathbb{N}$. For the adjacency matrix A, it is known that $||A||_1 = \deg_{\max}$. Furthermore, $||A||_2 = \sigma_{\max}$.

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 Equation (2). Several approximation algorithms have been developed to decrease the practical running times of this formulation (e.g., based on low-rank approximation [1]). Nathan et al. [19] prove a relationship between the numerical approximation quality of Equation (2) and the resulting Katz ranking quality. Although 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. [9] 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 work from Nathan and Bader [18] 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. Nathan and Bader give a global bound on the quality of their solution; however, we are able to compute per-node bounds that can guarantee the correctness of our ranking. Finally, Nathan and Bader'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.

Techniques introduced in the conference version of this article are used in the construction of a greedy optimization algorithm for the GED-Walk group centrality measure [2]. GED-Walk is related to Katz but differs in two key aspects: (i) it assigns a centrality score to sets of vertices and not to individual vertices, and (ii) it counts walks that cross vertices and not walks that start (or end) at certain vertices.

3 ITERATIVE IMPROVEMENT OF KATZ BOUNDS

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 lower and upper bounds, respectively, 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

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

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, we present the following definition.

Definition 3.1. In the same setting as before, let $\epsilon > 0$. We say that w and v are *ϵ-separated*, if and only if

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

Intuitively, the introduction of ϵ makes the ϵ-separation 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 $\epsilon > 0$.

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 ranking 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.

Top-k set criterion: Stop once the top-k nodes are ϵ-separated from all other nodes. This yields the set of top-k nodes without ranking the nodes within this set.

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.

3.1 Construction of Per-Node Bounds

For our algorithm to work, we need two additional properties from $\ell_r(v)$ and $u_r(v)$.

$$\lim_{r\to\infty} \ell_r(v) = \lim_{r\to\infty} u_r(v) = \mathbf{c}(v) \quad \text{for all } v \in V \tag{P1}$$

$$\ell_{r+1}(v) \geq \ell_r(v) \quad \text{and} \quad u_{r+1}(v) \leq u_r(v) \quad \text{for all } r \in \mathbb{N} \tag{P2}$$

(P1) ensures that we can satisfy our stopping criteria in a finite number of steps: if ℓ_r and u_r converge to the true values of $c(v)$, all vertices will be ϵ-separated after a finite number of iterations. However, (P2) (i.e., the property that ℓ_r and u_r are monotone in r) guarantees that the algorithm makes progress toward achieving ϵ-separation in each iteration. In particular, if (P2) holds, vertices that are ϵ-separated in some iteration of the algorithm will also be ϵ-separated in all future iterations (i.e., also after r is increased).

We now construct ℓ_r and u_r. First, we notice that a simple lower bound on the Katz centrality of a node v can be obtained by truncating the series in Equation (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.

To get an appropriate upper bound, we will prove a series of statements on $\sum_{i=r+1}^{\infty} \omega_i(v)$.

[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.

Lemma 3.1. *Let $C, \gamma > 0$ be constants. If $||A^i||_1 \leq C\gamma^i$ and $\alpha < 1/\gamma$, then one obtains a bound of*

$$\sum_{i=r+1}^{\infty} \alpha^i \omega_i(x) \leq \alpha^{r+1} \omega_r(x) C \frac{\gamma}{1-\alpha\gamma}$$

on the tail of Katz centrality.

Values of C and γ will later be chosen appropriately to obtain a tangible bound.

Proof. $\omega_i(x)$ can be restated as a ℓ_1-norm in $\mathbb{R}^n$: $\omega_i(x) = ||A^i e_x||_1$. Here, e_x denotes the x-th standard unit vector in $\mathbb{R}^n$. Using the fact that the induced ℓ_1-norm is compatible with the ℓ_1-norm on $\mathbb{R}^n$, we can see that $\omega_{r+i}(x) = ||A^{r+i} e_x||_1 \leq ||A^i||_1 \, ||A^r e_x||_1 = ||A^i||_1 \, \omega_r(x)$. Hence, it holds that

$$\sum_{i=r+1}^{\infty} \alpha^i \omega_i(x) \leq \sum_{i=1}^{\infty} \alpha^{r+i} ||A^i||_1 \, \omega_r(x) \leq \alpha^r \omega_r(x) C \sum_{i=1}^{\infty} (\alpha\gamma)^i.$$

Note that the last term is a geometric series. As $\alpha\gamma < 1$, this series converges to $\alpha\gamma/(1-\alpha\gamma)$; inserting this expression yields the result of the lemma. □

The following lemma will help us prove that (P1) holds for bounds that are based on Lemma 3.1.

Lemma 3.2. *The bound from Lemma 3.1 converges:*

$$\lim_{r\to\infty} \alpha^{r+1} \omega_r(x) C \frac{\gamma}{1-\alpha\gamma} = 0.$$

Proof. It is enough to show that $\lim_{r\to\infty} \alpha^r \omega_r(x) = 0$. We use a strategy similar to the proof of Lemma 3.1. It holds that $\alpha^r \omega_r(x) \leq \alpha^r ||A^r e_x||_1 \leq \alpha^r C\gamma^r ||e_x||_1$. The fact that $\alpha\gamma < 1$ completes the proof. □

3.2 Spectral Bound

We can now derive a bound on $\sum_{i=r+1}^{\infty} \omega_i(v)$ for arbitrary α by finding appropriate values for C and γ.

Corollary 3.3. *For general α (i.e., $\alpha < 1/\sigma_{\max}$), the tail of Katz centrality is bounded as follows:*

$$\sum_{i=r+1}^{\infty} \alpha^i \omega_i(x) \leq \alpha^{r+1} \omega_r(x) \sqrt{n} \, \frac{\sigma_{\max}}{1-\alpha\,\sigma_{\max}}.$$

Proof. Due to Lemma 3.1, it is enough to find an appropriate bound of $||A^i||_1$. Indeed, using the fact that $||M||_1 \leq \sqrt{n}\,||M||_2$ for arbitrary $n \times n$ matrices M (e.g., see the work of Feng and Tonge [8]), we can see that

$$||A^i||_1 \leq \sqrt{n}\,||A^i||_2 \leq \sqrt{n}\,||A||_2^i = \sqrt{n}\,\sigma_{\max}^i.$$

Here, the last equality uses the fact that the induced 2-norm is equal to the largest singular value. Note that according to Lemma 3.1, this bound is valid for all $\alpha < 1/\sigma_{\max}$ (i.e., for all α for which the Katz centrality exists). □

Unfortunately, the bound from Corollary 3.3 is not necessarily monotone in r—that is, (P2) does not necessarily hold. Since we want that property to hold for u_r, we define

$$u_r^{\text{spec}}(x) := \max_{i\in\{1,\ldots,r\}} \sum_{i=1}^{r} \alpha^r \omega_r(x) + \alpha^{r+1} \omega_r(x) \sqrt{n} \, \frac{\sigma_{\max}}{1-\alpha\,\sigma_{\max}}$$

to realize u_r in the general case (i.e., $\alpha < \sigma_{\max}$).

3.3 Combinatorial Bound

In special cases, we can improve upon u_r^{spec}: although Corollary 3.3 answers the question of a suitable bound on the tail of Katz centrality in full generality (i.e., for all possible values of α), it requires a computation of $\sigma_{\max}$. This computation is comparable in complexity to computing the Katz centrality itself; hence, it is desirable to find bounds that can be computed with less overhead.[4]

Corollary 3.4. *For $\alpha < 1/\deg_{\max}$, the tail of Katz centrality is bounded by*

$$\sum_{i=r+1}^{\infty} \alpha^i \omega_i(x) \leq \alpha^{r+1} \omega_r(x) \frac{\deg_{\max}}{1 - \alpha \deg_{\max}}.$$

Proof. Again, it is possible to use Lemma 3.1 and the fact that $||A||_1 = \deg_{\max}$ to show the corollary. For a combinatorial proof of the corollary, it is enough to see that $\omega_{i+1}(x) \leq \deg_{\max} \omega_i(x)$. This is true because any walk of length i that starts at x can be extended to at most $\deg_{\max}$ paths of length $i+1$ starting at x (by concatenating an edge to the end of the walk). The corollary now follows by applying induction and reusing the strategy of the proof of Lemma 3.1. □

Based on Corollary 3.4, we define the following bound to realize u_r:

$$u_r^{\text{comb}}(x) := \sum_{i=1}^{r} \alpha^r \omega_r(x) + \alpha^{r+1} \omega_r(x) \frac{\deg_{\max}}{1 - \alpha \deg_{\max}}.$$

It is worth remarking that graphs exist for which the bound from Corollary 3.4 is tight.

Lemma 3.5. *If G is a complete graph, $u_i(x) = \mathbf{c}(x)$ for all $x \in V$ and $i \in \mathbb{N}$.*

Proof. Consider the complete graph with n vertices. Then, $\omega_i(x) = (n-1)^i$ for all $x \in V$. Let $\delta > 0$ be a constant such that $\alpha = \frac{\delta}{\deg_{\max}} = \frac{\delta}{n-1}$. Equation (1) does not converge for $\delta \geq 1$. However, for $\delta < 1$, the Katz centrality is given by $\mathbf{c}(x) = \frac{\delta}{1-\delta}$. A short calculation (i.e., rewriting the partial sum of the geometric series in $u_i(x)$) shows that Corollary 3.4 yields the upper bound $u_i^{\text{comb}}(x) = \mathbf{c}(x)$ for all $i \in \mathbb{N}$ and $x \in V$. □

Bound u^{comb} is monotone—that is, it satisfies (P2).

Lemma 3.6. *For each $x \in V$, $u_i^{\text{comb}}(x)$ is non-increasing in i.*

Proof. We prove this lemma in slightly more general form. In particular, consider the case where Lemma 3.1 holds for $C = 1$. In this case, we can derive a bound of the form $u_r(x) = \sum_{i=1}^{r} \alpha^i \omega_i(x) + \alpha^{r+1} \omega_r(x) \frac{\gamma}{1-\alpha\gamma}$. It holds that

$$\begin{aligned} u_{i+1} - u_i(x) &= \alpha^{i+1} \omega_{i+1}(x) + \alpha^{i+2} \omega_{i+1}(x) \frac{\gamma}{1-\alpha\gamma} - \alpha^{i+1} \omega_i(x) \frac{\gamma}{1-\alpha\gamma} \\ &= \alpha^{i+1} \left(\frac{1-\alpha\gamma}{\gamma} + \alpha \right) \omega_{i+1}(x) \frac{\gamma}{1-\alpha\gamma} - \alpha^{i+1} \omega_i(x) \frac{\gamma}{1-\alpha\gamma} \\ &= \alpha^{i+1} \left(\frac{1}{\gamma} \omega_{i+1}(x) - \omega_i(x) \right) \frac{\gamma}{1-\alpha\gamma}. \end{aligned}$$

[4] In fact, 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 the number of iterations.

ALGORITHM 1: Katz centrality bound computation for static graphs

```
Initialize c_0(x) ← 0   ∀x ∈ V
Initialize r ← 0 and ω_0(x) ← 1   ∀x ∈ V
Initialize set of active nodes: M ← V
while not converged() do
    Set r ← r + 1 and ω_r(x) ← 0   ∀x ∈ V
    for all v ∈ V do
        for all v → u ∈ E do
            ω_r(v) ← ω_r(v) + ω_{r−1}(u)
        c_r(v) ← c_{r−1}(v) + α^r ω_r(v)
        Compute ℓ_r(v) and u_r(v)
function converged()
    partialSort(M, k, ℓ_r, decreasing)
    for all i ∈ {k + 1, . . . , |V|} do
        if u_r(M[i]) − ε < ℓ_r(M[k]) then
            M ← M \ {v}
    if |M| > k then
        return false
    for all i ∈ {2, . . . , min(|M|, k)} do
        if u_r(M[i]) − ε ≥ ℓ_r(M[i − 1]) then
            return false
    return true
```

Thus, it is enough to show that $\frac{1}{\gamma}\omega_{i+1}(x) - \omega_i(x) \leq 0$. The latter inequality follows by rewriting $\omega_i(x)$ as a 1-norm and using the properties of γ (i.e., Lemma 3.1):

$$\frac{1}{\gamma}\omega_{i+1}(x) = \frac{1}{\gamma}||A^{i+1}e_x||_1 \leq \frac{1}{\gamma}||A||_1||A^i e_x||_1 \leq ||A^i e_x|| = \omega_i(x).$$

Lemma 3.6 follows for $\gamma = \deg_{\max}$. □

3.4 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,

$$\omega_i(v) = \sum_{v \to x \in E} \omega_{i-1}(x). \tag{4}$$

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 section. 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 3.6, 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 Equation (4). 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).

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 afterward checks if the remaining top-k nodes are correctly ordered. These checks are performed by testing if the ϵ-separation condition from Equation (3) is true.

Complexity analysis. The sequential worst-case time complexity of Algorithm 1 is $O(r\,|E| + r\,\mathcal{P})$, where r is the number of iterations and $\mathcal{P}$ 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 $O(r\,\frac{|V|}{p}\,\deg_{max} + r\,\mathcal{P})$ on a parallel machine with p processors. The complexity of CONVERGED, the top-k ranking convergence criterion, is dominated by the $O(|V| + k \log k)$ complexity of partial sorting. Both the score and the pair criteria can be implemented in $O(1)$. Thus, to determine the overall complexity, it remains necessary to compute the value of r in the worst case.

LEMMA 3.7. *Using the bound of Lemma 3.1, the running time complexity of the algorithm is in* $O(\frac{\log(C\gamma/\epsilon)}{\log(1/(\alpha\gamma))}\,(|V| + |E|))$.

PROOF. The algorithm terminates when all pairs of vertices $(x, x') \in V \times V$ are ϵ-separated. Without loss of generality, let $\ell_r(x) \leq \ell_r(x')$. Then, x and x' are ϵ-separated once $u_r(x) - \ell_r(x') < \epsilon$ and it is sufficient to determine r such that $u_r(x) - \ell_r(x') \leq u_r(x) - \ell_r(x) < \epsilon$. By plugging in Lemma 3.1, this can be rewritten to $\alpha^{r+1}\omega_r(x)C\frac{\gamma}{1-\alpha\gamma} < \epsilon$. By the definition of C and γ, it holds that $\omega_r(x) \leq C\gamma^r$ and the left-hand side of this inequality is smaller than or equal to $C^2\frac{(\alpha\gamma)^{r+1}}{1-\alpha\gamma}$. Using the fact that $\alpha\gamma < 1$, a straightforward calculation now shows that for all $r > \frac{\log(\alpha C^2\gamma/(\epsilon(1-\alpha\gamma)))}{\log(1/(\alpha\gamma))}$, it holds that $\alpha^{r+1}\omega_r(x)C\frac{\gamma}{1-\alpha\gamma} \leq C^2\frac{(\alpha\gamma)^{r+1}}{1-\alpha\gamma} < \epsilon$. □

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, the heuristic of Foster et al. 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. Adding new (i.e., isolated) nodes to the graph does not affect Katz centrality; these new nodes receive a centrality score of zero. Likewise, removing isolated nodes from the graph does not affect the score of any remaining node. Adding (or removing) non-isolated nodes is handled by adding (or removing) isolated nodes followed by edge insertions (or preceded by edge deletions).

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 depicts the pseudocode of our dynamic Katz algorithm. $\mathcal{I}$ denotes the set of edges to be inserted, whereas $\mathcal{D}$ 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)**

ALGORITHM 2: Dynamic Katz update procedure

$E \leftarrow E \setminus \mathcal{D}$
$S \leftarrow \emptyset, T \leftarrow \emptyset$
for all $w \to v \in \mathcal{I} \cup \mathcal{D}$ **do**
  $S \leftarrow S \cup \{w\}$
  $T \leftarrow T \cup \{v\}$
for all $i \in \{1, \ldots, r\}$ **do**
  updateLevel(i)
for all $w \in S$ **do**
  Recompute $\ell_r(w)$ and $u_r(w)$ from $\mathbf{c}_r(w)$
for all $w \in V$ **do**
  if $u_r(w) \geq \min_{x \in M} \ell_r(x) - \epsilon$ **then**
    $M \leftarrow M \cup \{w\}$ ▷ Reactivation
$E \leftarrow E \cup \mathcal{I}$
while not converged() **do**
  Run more iterations of static algorithm

procedure updateLevel(i)
  for all $v \in S \cup T$ **do**
    $\omega'_i(v) \leftarrow \omega_i(v)$
  for all $v \in S$ **do**
    for all $w \to v \in E$ **do**
      $S \leftarrow S \cup \{w\}$
      $\omega'_i(w) \leftarrow \omega'_i(w) - \omega_{i-1}(v) + \omega'_{i-1}(v)$
  for all $w \to v \in \mathcal{I}$ **do**
    $\omega'_i(w) \leftarrow \omega'_i(w) + \omega'_{i-1}(v)$
  for all $w \to v \in \mathcal{D}$ **do**
    $\omega'_i(w) \leftarrow \omega'_i(w) - \omega_{i-1}(v)$
  for all $w \in S$ **do**
    $\mathbf{c}_i(w) \leftarrow \mathbf{c}_i(w) - \alpha^i \omega_i(w) + \alpha^i \omega'_i(w)$

through the reverse graph of G and updates ω_i for all nodes nodes that were reached in steps 1 through 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* afterward: 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 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 $O(r|E| + C)$, where 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 the entries of this vector, as this value is required for iteration $i + 1$. For readability, we omit this modification in the pseudocode.

5 EXPERIMENTS

Implementation details. The new algorithm in this article 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

Table 1. Details of Instances Used in the Experimental Section

Name	Origin	$\|V\|$	$\|E\|$	diam
roadNet-PA	Road	1,088,092	1,541,898	794
roadNet-CA	Road	1,965,206	2,766,607	865
cit-Patents	Citation	3,774,768	16,518,948	26
soc-pokec-relationships	Social	1,632,803	30,622,564	14
com-lj	Social	3,997,962	34,681,189	21
dimacs10-uk-2002	Link	23,947,347	57,708,624	45
sx-stackoverflow	Q&A	2,601,977	63,497,050	11
soc-LiveJournal1	Social	4,847,571	68,993,773	20
com-orkut	Social	3,072,441	117,185,083	10
com-friendster	Social	65,608,366	437,217,424	37
twitter	Social	41,652,230	1,468,365,182	23
wikipedia_link_en	Link	13,593,032	1,806,067,135	12

NetworKit [21]; on the GPU, we use Hornet.[5] The Hornet data structure is architecture independent, although 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. The library contains parallel shared-memory implementations of many popular graph algorithms and can handle networks with billions of edges.

The Hornet [7], an efficient extension to the cuSTINGER [10] 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 10 times faster.

In our experiments, we compare our new algorithm to the heuristic of Foster et al. and a **conjugate gradient (CG)** algorithm (without preconditioning) that solves Equation (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. These experiments use the Simexpal [3] toolkit to ensure reproducibility of the results. The networks originate from diverse real-world applications and were taken from SNAP [16] and KONECT [13]. Details about the exact instances that we used can be found in Table 1. 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. In Table 1, $|V|$ and $|E|$ refer to the number of vertices and edges in the original data (before the transformation to undirected graphs).

Most CPU experiments ran on a machine with dual-socket Intel Xeon E5-2690 v2 CPUs with 10 cores per socket[6] and 128 GiB RAM; the experiments on the spectral bound (Section 5.2) ran sequentially on a Intel Xeon Gold 6154 CPU. Our GPU experiments are conducted on an NVIDIA

[5]Hornet can be found at https://github.com/hornet-gt, whereas NetworKit is available from https://github.com/networkit/networkit. Both projects are open source, including the implementations of our new algorithm.
[6]Hyperthreading was disabled for the experiments.

Table 2. 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.

P100 GPU that has 56 streaming multiprocessors and 64 **streaming processors (SPs)** per streaming multiprocessor (for a total of 3,584 SPs) and has 16 GB of HBM2 memory. To effectively use the GPU, the number of active threads need to be roughly eight 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.

Unless stated otherwise, we use the combinatorial bound and $\alpha = \frac{1}{\deg_{\max}+1}$ in our experiments.

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 the heuristic of Foster et al. We run CG until the 2-norm of 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 the heuristic of Foster et al., 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 2 summarizes the results of the evaluation. The fourth column of Table 2 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 2 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 10 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 Foster et al. algorithms on all instances. The algorithm of Foster et al. is faster than our algorithm for $\epsilon = 10^{-5}$ on 3 out of 10 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.

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. To demonstrate that our

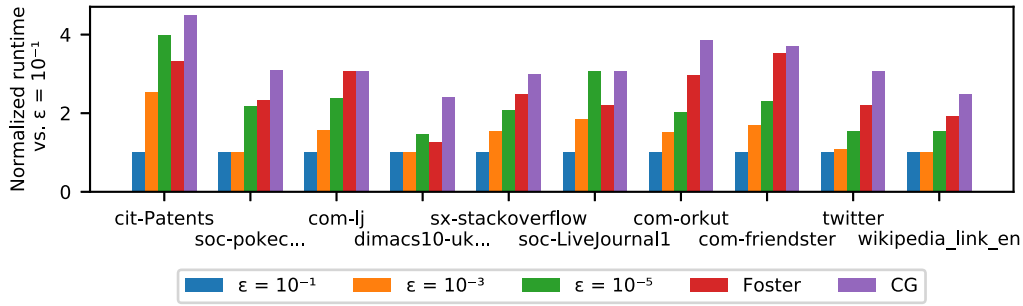

Fig. 1. Katz performance on individual instances.

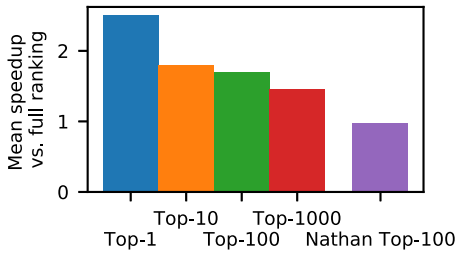

Fig. 2. Top-k speedup over full ranking.

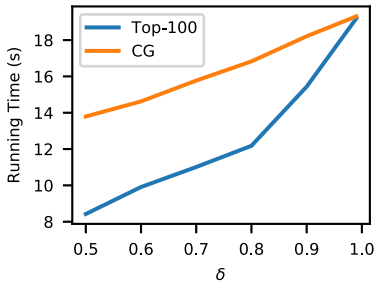

Fig. 3. Performance of the spectral bound.

algorithm performs well over a large range of k, we pick k between 1 and 1,000, increasing in factors of 10. The figure also includes the approach of Nathan et al. [19], who conducted experiments on real-world graphs and concluded that solving Equation (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 $k \leq 1{,}000$. We note that the running times can still be improved considerably if the top-k set criterion is used instead of the top-k ranking criterion. For example, for $k = 100$, the running time improves by a factor of 1.4× in the geometric mean, whereas for $k = 1{,}000$, it improves by 1.3×.

Finally, we investigate how the performance of our algorithm is affected by the characteristics of the input graph. We compare the performance of our top-k algorithm for $k = 100$ on road and non-road networks. We pick $\epsilon = 10^{-15}$. For roadNet-CA and roadNet-PA, we get speedups of 1.94× and 1.78× over CG, respectively. On other networks, we get a geometric mean speedup of 1.51×. Overall, our algorithm performs well on both high-diameter and low-diameter networks. However, both algorithms require more iterations on high-diameter networks than on low-diameter networks: CG requires 3.5× more iterations on the two road networks, whereas our algorithm requires only 2.2× more iterations.

5.2 Performance of Spectral Bound

Next, we analyze the performance of our algorithm when α approaches the largest valid value of $\frac{1}{\sigma_{\max}}$. More precisely, we run our top-100 algorithm and the CG algorithm with $\alpha = \frac{\delta}{\sigma_{\max}}$, for values of δ between 0.5 and 0.99. The results are depicted in Figure 3. Our algorithm is faster than CG for $\delta < 0.99$. CG scales better as δ approaches 1; this is expected since CG picks the direction in which

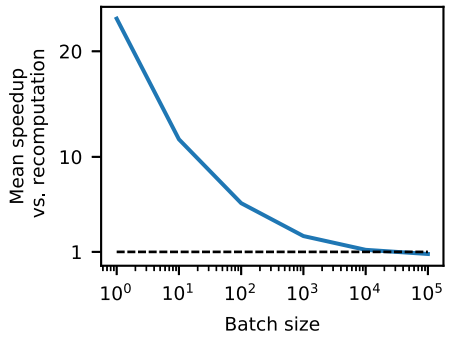

Fig. 4. Dynamic update performance.

the solution is improved in a more sophisticated way (however, in contrast to our algorithm, it does not make direct guarantees about the correctness of the ranking). We remark that both algorithms run into numerical stability issues if δ increases beyond the range from Figure 3 (i.e., for $\delta \geq 0.999$). This is not surprising as Katz centrality diverges for $\delta \to 1$.

5.3 Evaluation of the Dynamic Katz Algorithm

In our next experiment, we evaluate the performance of our dynamic Katz algorithm to compute top-1,000 rankings using $\epsilon = 10^{-4}$. The experiment is performed on the graphs of Table 1. 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 although we only show results for edge deletion, edge insertion is completely symmetric in Algorithm 2.

Figure 4 summarizes the results of the experiment. We present the geometric mean speedup over all instances. For batch sizes $b \leq 1{,}000$, 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 ($b > 10{,}000$), 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.

5.4 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 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-10, 000 convergence criterion with $\epsilon = 10^{-6}$.

Figure 5 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. The parallel CPU configurations yield moderate speedups over the sequential implementation; however, the GPU gives a significant speedup over the 16-core CPU configuration.[7] Even compared to a 20-core CPU configuration (not depicted in the plots; Table 3), the GPU achieves a (geometric) mean speedup of 10×.

[7]Our CPU implementation uses a sequential algorithm for partial sorting while the GPU sorts in parallel.

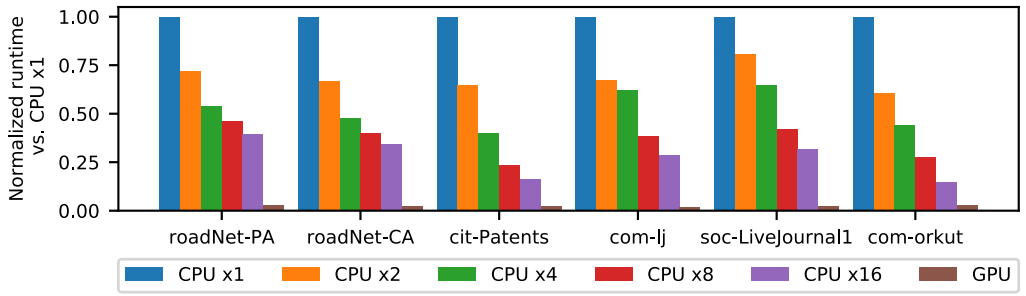

Fig. 5. Scalability of parallel CPU and GPU implementations.

Table 3. Absolute Running Times and Parallel Speedups (Versus Sequential CPU Baseline) of CPU- and GPU-Based Codes for GPU Instances

		Number of Cores					
Name	Sequential	2	4	8	16	20	GPU
roadNet-PA	670 ms	1.39×	1.85×	2.16×	2.55×	2.18×	33.50×
roadNet-CA	1018 ms	1.50×	2.09×	2.51×	2.93×	2.69×	40.72×
cit-Patents	4751 ms	1.55×	2.52×	4.23×	6.19×	6.41×	41.68×
com-lj	3916 ms	1.48×	1.60×	2.61×	3.51×	4.07×	56.75×
soc-LiveJournal1	3710 ms	1.24×	1.55×	2.39×	3.14×	4.40×	42.64×
com-orkut	8205 ms	1.65×	2.26×	3.64×	6.69×	6.94×	38.52×

Note: Data is presented for all instances of our GPU experiments. We report data on a single run of each algorithm (preliminary experiments demonstrated that the deviations among different runs are generally below 10% of the mean).

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 article, we presented an algorithm for Katz centrality that computes upper and lower bounds on the Katz score of individual nodes. Experiments demonstrated that our algorithm is able to compute highly accurate Katz centrality rankings quickly. It outperforms both linear algebra formulations and heuristics, with speedups between 150% and 350% depending on the desired correctness guarantees.

Future work could try to apply the ϵ-separation framework to other ranking problems. It would also be interesting to either prove that there are graphs where the spectral bound is tight (indicating that our current algorithm cannot be further improved), or to derive stricter per-node bounds for Katz centrality. On the implementation side, our new algorithm could be formulated in the language of GraphBLAS [12] to enable it to run on a variety of upcoming software and hardware architectures.

REFERENCES

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

[2] Eugenio Angriman, Alexander van der Grinten, Aleksandar Bojchevski, Daniel Zügner, Stephan Günnemann, and Henning Meyerhenke. 2020. Group centrality maximization for large-scale graphs. In *Proceedings of the SIAM Symposium on Algorithm Engineering and Experiments (ALEXNEX'20)*. 56–69.

[3] Eugenio Angriman, Alexander van der Grinten, Moritz von Looz, Henning Meyerhenke, Martin Nöllenburg, Maria Predari, and Charilaos Tzovas. 2019. Guidelines for experimental algorithmics: A case study in network analysis. *Algorithms* 12, 7 (2019), 127.

[4] Elisabetta Bergamini, Michele Borassi, Pierluigi Crescenzi, Andrea Marino, and Henning Meyerhenke. 2018. Computing top-k closeness centrality faster in unweighted graphs. In *Proceedings of the 2016 18th Workshop on Algorithm Engineering and Experiments (ALENEX'18)*. 68–80. https://doi.org/10.1137/1.9781611974317.6

[5] Patrick Bisenius, Elisabetta Bergamin, Eugenio Angriman, and Henning Meyerhenke. 2018. Computing top-k closeness centrality in fully-dynamic graphs. In *Proceedings of the 2018 20th Workshop on Algorithm Engineering and Experiments (ALENEX'18)*. 21–35. https://doi.org/10.1137/1.9781611975055.3

[6] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems* 30, 1 (1998), 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X

[7] F. Busato, O. Green, N. Bombieri, and D. A. Bader. 2018. Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs. In *Proceedings of the IEEE Conference on High Performance Extreme Computing (HPEC'18)*.

[8] Bao Qi Feng and Andrew Tonge. 2007. Equivalence constants for certain matrix norms II. *Linear Algebra and Its Applications* 420, 2 (2007), 388–399. https://doi.org/10.1016/j.laa.2006.07.015

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

[10] O. Green and D. A. Bader. 2016. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *Proceedings of the IEEE Conference on High Performance Embedded Computing Workshop (HPEC'16)*.

[11] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (March 1953), 39–43. https://doi.org/10.1007/BF02289026

[12] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, et al. 2016. Mathematical foundations of the GraphBLAS. In *Proceedings of the 2016 IEEE High Performance Extreme Computing Conference (HPEC'16)*. 1–9. https://doi.org/10.1109/HPEC.2016.7761646

[13] Jérôme Kunegis. 2013. KONECT: The Koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web (WWW'13 Companion)*. ACM, New York, NY, 1343–1350. https://doi.org/10.1145/2487788.2488173

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

[15] Min-Joong Lee and Chin-Wan Chung. 2014. Finding K-highest betweenness centrality vertices in graphs. In *Proceedings of the 23rd International Conference on World Wide Web (WWW'14 Companion)*. ACM, New York, NY, 339–340. https://doi.org/10.1145/2567948.2577358

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

[17] Eisha Nathan and David A. Bader. 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 (ASONAM'17)*. ACM, New York, NY, 149–154. https://doi.org/10.1145/3110025.3110034

[18] Eisha Nathan and David A. Bader. 2018. Approximating personalized Katz centrality in dynamic graphs. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski (Eds.). Springer International Publishing, Cham, Switzerland, 290–302.

[19] Eisha Nathan, Geoffrey Sanders, James Fairbanks, Van Emden Henson, and David A. Bader. 2017. Graph ranking guarantees for numerical approximations to Katz centrality. *Procedia Computer Science* 108 (2017), 68–78. https://doi.org/10.1016/j.procs.2017.05.021

[20] Mark Newman. 2010. *Networks: An Introduction*. Oxford University Press, New York, NY.

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

[22] Alexander van der Grinten, Elisabetta Bergamini, Oded Green, David A. Bader, and Henning Meyerhenke. 2018. Scalable Katz ranking computation in large static and dynamic graphs. In *Proceedings of the 2018 European Symposium on Algorithms (ESA'18)*, Vol. 112. Article 42, 14 pages.

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

Received December 2020; revised November 2021; accepted March 2022