NUMERICAL AND STREAMING ANALYSES OF CENTRALITY MEASURES ON GRAPHS

A Dissertation
Presented to
The Academic Faculty

By

Eisha R. Nathan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology

May 2018

NUMERICAL AND STREAMING ANALYSES OF CENTRALITY MEASURES ON GRAPHS

Approved by:

Dr. David A. Bader, Advisor
Computational Science and Engineering
Georgia Institute of Technology

Dr. Srinivas Aluru
Computational Science and Engineering
Georgia Institute of Technology

Dr. Umit Catalyurek
Computational Science and Engineering
Georgia Institute of Technology

Dr. Bistra Dilkina
Computer Science Department
University of Southern California

Dr. Jason Riedy
Computational Science and Engineering
Georgia Institute of Technology

Dr. Geoffrey Sanders
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

Date Approved: March 14, 2018

To my grandparents,

the two I have in this world,

and the two I know are watching me everyday from above.

ACKNOWLEDGEMENTS

I would like to thank my advisor, David Bader, for introducing me to the world of graphs, and for his mentorship over the last four years. A very important thank you to Geoff Sanders for not only being on my thesis committee, but for being an amazing mentor with constant support and ideas along the way, and thanks to Van Henson for allowing me to intern at LLNL multiple summers, introducing me to a whole network of brilliant researchers.

Thanks to Jason Riedy for helpful guidance, always being ready to engage in interesting research (or otherwise) talks with me, brainstorm new research directions, and for serving on my committee. Thanks to the rest of my committee, Bistra Dilkina, Umit Catalyurek, and Srinivas Aluru, as well, for their time and feedback.

Thanks to my friends both inside and outside the department for keeping me (moderately) sane these last four years: Debolina Dasgupta, Amrita Gupta, James Fairbanks, Sabra Neal, Vinh Nguyen, Colin Ponce, and Anita Zakrzewska. Each of you has made the last few years so much more bearable.

Last, the most important acknowledgment goes to my family. Thanks especially to my parents, Ram and Saumya Nathan, and sister Anusha, for encouraging me every step of the way and motivating me to never give up. I am so lucky to have these three in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Graph data represent information about entities (vertices) and the relationships or connections between them (edges). In real-world networks today, new data are constantly being produced, leading to the notion of dynamic graphs. When analyzing large graphs, a common problem of interest is to identify the most important vertices in a graph. Vertex importance is calculated using centrality, where a centrality metric assigns a value to each vertex in the graph and these values can then be turned into rankings. This dissertation presents novel advances in the field of graph analysis by providing numerical and streaming techniques that help us better understand how to compute centrality measures. Several centrality measures are calculated by solving a linear system but since these linear systems are large, iterative solvers are often used as an alternate method to approximate the solution. We relate the two research areas of numerical accuracy and data mining by understanding how the error in a solver affects the relative ranking of vertices in a graph. To calculate the centrality values of vertices in a dynamic graph, the most naive method is to recompute the scores from scratch every time the graph is changed, but as the graph size grows larger this recomputation is computationally infeasible. We present four dynamic algorithms for updating different centrality metrics in evolving networks. All dynamic algorithms are more efficient than their static counterparts while maintaining good quality of the centrality scores. This dissertation concludes by applying methods discussed for the computation of centrality metrics to community detection, and we present a new algorithm for identifying local communities in a dynamic graph using personalized centrality.

CHAPTER 1
INTRODUCTION

Graph data represent information about entities (vertices) and the relationships or connections between them (edges). In real-world networks today, new data are constantly being produced, leading to the notion of dynamic graphs. Dynamic graph data can represent the changing relationships in social networks, financial transaction networks, and computer networks. For example, in a Facebook graph, a vertex could represent a person where the addition or deletion of an edge would represent a friendship being created or removed, respectively. When analyzing large graphs, a common problem of interest is to identify the most important vertices in a graph [1]. Vertex importance is calculated using centrality metrics, where a centrality metric assigns a value to each vertex in the graph. These values can then be turned into rankings indicating relative importance. This dissertation presents novel advances in the field of graph analysis by providing techniques that help us better understand how to compute centrality measures on graphs. These techniques broadly fall under two categories: (1) numerical and (2) dynamic analysis of centrality measures.

Calculations of several centrality measures involve solving a linear system, where the solution is an n-length vector of values indicating centrality values for all n vertices in the graph. Oftentimes these linear systems are large and computationally expensive to solve directly and exactly so iterative solvers are used as an alternate method to approximate the solution to the system. Since iterative solvers produce only an approximation to the exact solution, there is inherently some error in the obtained ranking vector. Without knowing the exact solution, it is generally impossible to determine how the numerical error in the approximation vector from the iterative solver affects the resulting relative ranking of vertices in the graph. Chapter 3 addresses

this issue and relates the two research areas of numerical accuracy and data mining by understanding how the error in a solver affects the accuracy of ranking vertices in a graph [2, 3]. Current methods to identify the most highly ranked vertices in a graph run an iterative method to a high tolerance (most commonly machine precision) and return the highly ranked vertices as those with the highest centrality scores (or the vertices corresponding to the scores at the top of a sorted centrality vector). However, these methods are problematic because running a solver to a high accuracy such as machine precision can require many iterations to converge and therefore be very time and resource consuming. Furthermore, even though iterating to a high accuracy such as machine precision likely outputs an approximate centrality vector close to the unknown exact solution, there is currently no means to determine exactly how accurate the approximation is. This translates to not knowing if the highly ranked vertices returned (judged purely on the centrality scores in a sorted ranking vector) are actually the highly ranked ones with respect to the unknown exact solution's ranking. Our work addresses this problem. We study two walk-based centrality measures, Katz Centrality [4] and PageRank [5], both of which generally rank vertices by counting the number of walks of different lengths starting at each vertex in the graph and penalize longer walks with a user-chosen parameter. Bounding the error of the approximation vector compared to the exact solution (in the numerical problem) allows us to develop theory to guarantee relative ranking of vertices in graphs (in the data mining problem). Theory is proven and discussed for both Katz Centrality and PageRank. The theory presented lends itself to the development of a new stopping criterion for iterative methods that guarantees, upon termination at our new stopping criterion, returning the correct highly ranked vertices with respect to the unknown exact solution's ranking. Terminating at our new stopping criterion provides the previously missing theoretical guarantees of correctness of the highly ranked vertices and returns results significantly faster than the conventional method of iterating to a high

tolerance to identify the highly ranked vertices.

Switching from static to dynamic graph analysis, Chapter 4 presents dynamic algorithms for a variety of centrality measures. Several real-world datasets are comprised of temporal information and are therefore considered dynamic networks. To calculate the centrality values of vertices in a dynamic graph, the most naive method is to recompute the scores from scratch each time the graph is changed by taking snapshots of the graph over time and treating each snapshot as a separate static graph. This method quickly becomes problematic as the graph size grows larger and a pure static recomputation is rendered computationally infeasible. Furthermore, typically in large networks, updates to the graph will only affect a small portion of the graph. Recomputating from scratch everytime the graph is changed is both intractable and impractical. Therefore, when calculating analytics on dynamic graphs it is preferable to have analytics that update in real-time when the graph is changed without needing to perform a full static recomputation. First, we present two different algorithms for computing Katz Centrality scores in dynamic graphs. The first method presented in Section 4.1 studies the problem in a linear algebra-based environment [6, 7]. Phrasing the computation of Katz Centrality as the solution to a linear system allows the development of an algorithm that exploits properties of iterative solvers to quickly obtain an updated centrality vector for evolving networks. The second method presented is a non-linear algebraic algorithm tailored for calculating personalized Katz scores in graphs. When studying personalized scores, analysts may only be interested in calculating scores of vertices surrounding the seed vertex (with respect to which personalized scores are calculated) and be less interested in calculating exact scores of vertices far away from the seed and near the edge of the graph. A linear algebraic computation typically cannot distinguish between vertices close to the seed and those far away and so Section 4.2 presents an alternate, agglomerative dynamic algorithm for calculating personalized Katz scores in dynamic graphs [8]. Next, Section 4.3

presents a dynamic algorithm for calculating nonbacktracking walk-based centrality scores on the vertices of an evolving network. Katz Centrality scores are walk-based, where a walk allows for sequences with repetition of vertices and edges and assign centrality scores to vertices by weighting walks of different lengths. For example, a walk from vertices $0 \to 1 \to 0 \to 1 \to 0$ and a walk from vertices $0 \to 1 \to 2 \to 3 \to 4$ are weighted the same because they are the same length. While this measure suffices for some purposes, in other applications weighting walks of the same length equally fails. Consider the case of studying information diffusion in a network, where the walk between vertices $0 \to 1 \to 0 \to 1 \to 0$ is essentially useless and a walk that traverses more of the network such as $0 \to 1 \to 2 \to 3 \to 4$ ought to be given more weight. Therefore, we next study walks of a nonbacktracking nature, or specifically walks that do not allow the sequence from vertices $i \to j \to i$. Nonbacktracking walks can be used to calculate a centrality measure similar to Katz Centrality. Finally, moving back to a linear algebraic environment, Section 4.4 presents a dynamic algorithm for calculating matrix exponential-based centrality scores of vertices in an evolving network [9]. All dynamic algorithms presented in Chapter 4 are compared to their static counterparts: a pure static algorithm that recomputes the respective centrality metric from scratch everytime the graph is changed. Our methods contribute to the field with performance improvements (usually several orders of magnitude of speedup) while maintaining good quality of the centrality scores. We are able to reduce the computation time, and, when applicable, the number of iterations needed to converge to the solution in the dynamic setting compared to statically recomputing the scores. The quality of our methods never deteriorate over time for the examples shown in this dissertation, suggesting that they can be used for a large number of updates.

This dissertation concludes by applying methods discussed for the computation of centrality metrics to another popular graph analysis query: community detection.

Community detection is the task of identifying dense clusters, or closely related groups of vertices, in the graph. For example, in a Facebook network, communities may be groups of people who interact with each other fairly regularly or may be part of similar groups online. Typically, global community detection is performed by partitioning the network into smaller subgraphs where the members (vertices) of the subgraphs, or communities, are more closely linked to each other than to the rest of the network by some measure. Several metrics exist for quantifying how well connected vertices in a community are; most commonly this is done by some measure of the number of edges in between vertices inside the communities, or intra-community edges, compared to the number of edges leaving the community, or inter-community edges. An alternate definition identifies a community as individuals who have more *influence* on members of the same community than on individuals outside of the community, where influence is calculated through pairwise ranking of vertices in a graph. Local community detection is the task of identifying the community associated with a set of seed vertices of interest. Chapter 5 presents a new algorithm for identifying local communities in a dynamic graph using personalized centrality [10]. We explore the relationship between centrality and community detection to understand what a personalized centrality vector with respect to seed vertices can tell us about a local community associated with the same seed vertices. The techniques discussed in Chapter 4, Section 4.1 to update a centrality vector in a dynamic graph are extended to track local communities in dynamic graphs. Results on several synthetic networks (where ground truth is known) show that our method is able to accurately track the local communities with respect to seed vertices. Applying our method to real-world networks shows that we can identify similar quality communities to other commonly used community detection methods. Finally, the results of our analyses show that our dynamic algorithm is able to identify local communities in a fraction of the time it takes a corresponding static algorithm.

In summary, this thesis makes the following contributions:

- A new error bound on elements of a ranking vector to provide graph ranking guarantees to the computation of Katz Centrality and PageRank and demonstrations that this bound provides practical results in real datasets [2, 3]
- A new stopping criterion for iterative solvers to identify highly ranked vertices in a graph that reduces runtime compared to running a solver to machine precision [2, 3]
- Empirical evidence of a tighter probabilistic upper bound on $\|A\|_2$ compared to deterministic Gershgorin bounds for real-world graphs [2, 3]
- A new dynamic algorithm using linear algebra to update Katz Centrality scores in streaming graphs [6, 7]
- New agglomerative algorithms for approximating personalized Katz Centrality in both static and dynamic graphs [8]
- New algorithms for calculating nonbacktracking walk centrality scores of vertices in static and dynamic graphs
- A new algorithm for incrementally calculating exponential centrality in dynamic graphs [9]
- Development of a new method for identifying local communities using personalized centrality metrics [10]
- A new dynamic algorithm to identify local communities in evolving networks with validation on both synthetic and real-world dynamic graphs [10]

Together, the contributions in this thesis give us a better understanding of techniques that can be applied for the analysis of centrality measures on graphs.

CHAPTER 2
BACKGROUND AND LITERATURE REVIEW

This chapter provides an overview of the necessary concepts needed to understand the work presented in this dissertation. A brief overview of the basics of graph theory and linear algebra are presented in Sections 2.1 and 2.2 respectively, ranking methods on graphs are discussed in detail in Section 2.3, Section 2.4 surveys the literature in dynamic analysis of centrality measures in graphs, and Section 2.5 concludes with a discussion of literature on community detection.

2.1 Graph Theory

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

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

For undirected graphs, if there is an edge between vertices i and j, then there is a corresponding edge between vertices j and i, so $\forall i, j, A(i,j) = A(j,i)$. This dissertation assumes all graphs to be unweighted so all edge weights are 1, or $\forall i, j, A(i,j) = 1$. A dynamic graph changes over time due to edge insertions and/or deletions, as well as vertices being added or removed over time. As a graph changes, we can take snapshots of its current state at any time. We denote the snapshot of the dynamic graph G and its corresponding adjacency matrix at time t by $G_t = (V_t, E_t)$ and A_t. In our work, the vertex set stays the same over time so $\forall t, V_t = V$ and we deal only with edge insertions, although our work can easily be generalized to edge deletions. Changes to

the graph (and therefore its corresponding adjacency matrix) can be represented in matrix-form by an $n \times n$ change-matrix ΔA. If we insert edge (i, j) into the graph at time t, we set $\Delta A(i, j) = 1$. Similarly if we want to delete edge (i, j) we set $\Delta A(i, j) = -1$. A walk of length k in a graph is a series of k vertices $v_1, \cdots, v_k$ where both vertices and edges are allowed to repeat. A path is a walk where all vertices are distinct. Using powers of the adjacency matrix allows us to count walks in graphs, where $A^k(i, j)$ gives the number of walks of length k from vertex i to j [11].

Two commonly studied questions in graph analysis are (1) identifying important vertices and (2) identifying groups in graphs that vertices belong to. The first problem is answered through centrality, where a centrality metric provides a score for each vertex in the graph indicating its relative importance. A centrality metric can be represented by an $n \times 1$ vector where the ith value in the vector gives the value for the ith vertex in the graph. The second problem is answered through community detection, where a community can be broadly defined as a group of vertices more closely related to each other than the rest of the graph. We define a community of vertices as $C = \{v_1, \cdots, v_{|C|}\}$. This dissertation focuses primarily on the first question of centrality, but concludes with an application of methods to compute centrailty scores to the task of identifying communities in graphs.

2.2 Linear Algebra

Using linear algebra as a tool to aid in graph processing is a common theme in dealing with many algorithmic applications. In this section, we give a brief overview of linear algebra terminology used in the rest of the document.

An eigenvector of A is a vector $\mathbf{x}$ such that $A\mathbf{x}$ is parallel to $\mathbf{x}$. Mathematically we write this as $A\mathbf{x} = \lambda\mathbf{x}$ for some real or complex number λ. The number λ is called an eigenvalue of A associated with the eigenvector $\mathbf{x}$. A singular value σ and associated singular vectors $\mathbf{u}, \mathbf{v}$ are a nonnegative real-valued scalar and two vectors

such that $A\mathbf{v} = \sigma\mathbf{u}$ and $A^H\mathbf{u} = \sigma\mathbf{v}$, where A^H is the Hermitian transpose of A, or the complex conjugate transpose. The spectral radius $\rho(A)$ of A is given by the largest eigenvalue, $max_i|\lambda_i|$ and the matrix 2-norm $\|A\|_2$ is given by the largest singular value, σ_{max}. For the case of undirected graphs, $A^T = A$ and $\rho(A) = \|A\|_2$. By [12], $\rho(A) = \|A\|_2 \in [\sqrt{d_{max}}, d_{max}]$, where d_{max} is the maximum degree in the graph G.

Much of the work presented here seeks to come up with a solution to a numerical problem $M\mathbf{x} = \mathbf{b}$, where we aim to solve for $\mathbf{x}$ given M and $\mathbf{b}$. Solving this system exactly using direct methods is typically fairly computationally expensive, so iterative methods are used as an alternative to provide an approximation to $\mathbf{x}$. An iterative method starts with an initial guess $\mathbf{x}^{(0)}$ and iteratively improves the current guess with each iteration until reaching some stopping criterion. This stopping criterion can be a predetermined number of iterations, a desired level of accuracy, or some application-specific terminating criterion. Since iterative solvers are used to obtain approximations to the exact solution $\mathbf{x}^*$, at each step k of the iterative solver we denote the new approximation as $\mathbf{x}^{(k)}$.The error at each step is denoted as the difference between the exact and approximation, $\|\mathbf{x}^* - \mathbf{x}^{(k)}\|_2$ and the residual as $r_k = \|\mathbf{b} - M\mathbf{x}^{(k)}\|_2$, where $\|\cdot\|_2$ denotes the 2-norm. The residual at iteration k denotes how close the current approximation $\mathbf{x}^{(k)}$ is to solving the linear system. In practice as the exact solution is not known, typical stopping criteria for the solver use the residual, terminating when it hits a high accuracy. We use machine precision, or when $r_k \approx 10^{-15}$. All the work here assumes a starting approximation $\mathbf{x}^{(0)}$ as the all zeros vector, although in practice, any starting vector can be chosen.

The three iterative methods used in this dissertation are CONJUGATE GRADIENT, JACOBI, and GMRES (Generalized Minimal Residual Method), and are given in Algorithms 1, 2, and 3 respectively. Conjugate gradient is used if the matrix M is symmetric and GMRES is used for solving systems with unsymmetric matrices. In Algorithm 1, D is the matrix consisting of the diagonal entries from M and R is the

matrix of all off-diagonal entries of M. In Algorithm 3, the parameter k allows us to restart the GMRES algorithm every k iterations.

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

procedure JACOBI($M, \mathbf{b}, tol$)
  $k = 0$
  $\mathbf{x}^{(0)} = \mathbf{0}$
  $\mathbf{r}^{(0)} = \mathbf{b} - M\mathbf{x}^{(0)}$
  $D = diag(A)$
  $R = M - D$
  while $\|\mathbf{r}^{(k)}\|_2 > tol$ **do**
    $\mathbf{x}^{(k+1)} = D^{-1}(R\mathbf{x}^{(k)} + \mathbf{b})$
    $\mathbf{r}^{(k+1)} = \mathbf{b} - M\mathbf{x}^{(k+1)}$
    $k+ = 1$
  return $\mathbf{x}^{(k+1)}$

Algorithm 2 Solve $M\mathbf{x} = \mathbf{b}$ to tolerance tol using conjugate gradient algorithm.

procedure CONJUGATE_GRADIENT($M, \mathbf{b}, tol$)
  $\mathbf{x}^{(0)} = \mathbf{0}$
  $\mathbf{r}^{(0)} = \mathbf{b} - M\mathbf{x}^{(0)}$
  $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$
  $k = 0$
  while $\|\mathbf{r}^{(k)}\|_2 > tol$ **do**
    $\alpha_k = \frac{\mathbf{r}^{(k)T}\mathbf{r}^{(k)}}{\mathbf{p}^{(k)T}M\mathbf{p}^{(k)}}$
    $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k\mathbf{p}^{(k)}$
    $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k M\mathbf{p}^{(k)}$
    $\beta_k = \frac{\mathbf{r}^{(k+1)T}\mathbf{r}+\mathbf{1}^{(k)}}{\mathbf{r}^{(k)T}\mathbf{r}^{(k)}}$
    $\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} + \beta_k\mathbf{p}^{(k)}$
    $k+ = 1$
  return $\mathbf{x}^{(k+1)}$

2.3 Ranking in Graphs

One of the most popular questions arising from the analysis of large graphs is to determine the most important vertices in a graph. Vertex importance is referred to as *centrality*, and centrality scores can be used to provide *rankings* on the vertices of a graph. Section 2.3.1 gives a more in-depth background on several linear algebra

Algorithm 3 Solve $M\mathbf{x} = \mathbf{b}$ to tolerance tol using GMRES algorithm.

procedure GMRES($M, \mathbf{b}, tol, k$)
  $\mathbf{x}^{(0)} = \mathbf{0}$
  $\mathbf{r}^{(0)} = \mathbf{b} - M\mathbf{x}^{(0)}$
  $\mathbf{v}^{(1)} = \mathbf{r}^{(0)}/\|\mathbf{r}^{(0)}\|_2$
  while $\|\mathbf{r}^{(k)}\|_2 > tol$ **do**
    for $j = 1, 2, \cdots, k$ **do**
      $h_{ij} = (A\mathbf{v}^{(j)}, \mathbf{v}^{(i)})$
      $\widetilde{\mathbf{v}}^{(j+1)} = A\mathbf{v}^{(j)} - \sum_{i=1}^{j} h_{ij}\mathbf{v}^{(i)}$
      $h_{j+1,j} = \|\widetilde{\mathbf{v}}^{(j+1)}\|_2$
      $\mathbf{v}^{(j+1)} = \widetilde{\mathbf{v}}^{(j+1)}/h_{j+1,j}$
    Compute $\mathbf{y}^{(k)}$ s.t. $\mathbf{y}^{(k)}$ minimizes $\|\beta\mathbf{e}^{(1)} - \widetilde{H}_k\mathbf{y}^{(k)}\|_2$
    $\mathbf{x}^k = \mathbf{x}^{(0)} + V_m\mathbf{y}_k$
    $\mathbf{r}^{(k)} = \mathbf{b} - M\mathbf{x}^{(k)}$
  return $\mathbf{x}^{(k)}$

based metrics, and Section 2.3.2 provides background on the effect the approximation (using iterative solvers) of a centrality measure has on the final answer.

2.3.1 Functions of the adjacency matrix

Many centrality measures are obtained by solving a linear system on the adjancency matrix of the graph. The solution is a vector consisting of a number for each vertex in the graph identifying its relative importance. Obtaining an exact solution via direct methods is prohibitively computationally expensive, since we are typically required to take the inverse of a matrix. Although direct methods can usually obtain high accuracy solutions, these methods tend to consume large amounts of memory or take a long time to compute. For example, when graphs are small-world and scale-free (as are many real-world networks), direct methods like Cholesky require $\mathcal{O}(n^2)$ to $\mathcal{O}(n^3)$ computations [13]. In many real networks the amount of data is massive and n can be as large as millions or billions of vertices, so direct methods such as these do not scale and are impractical. Moreover, there is no computationally tractable technique to compute an exact solution for a general graph in finite precision arithmetic, so in practice, iterative methods are often used to obtain an approximate solution. Iterative

methods tend to use less memory than direct methods, where each iteration costs $\mathcal{O}(m)$, where m is the number of edges in the graph. However, in order for an iterative method to be cost effective, the number of iterations must be limited. Many real-world graphs are sparse and $m \ll n^2$ [14]. While occasionally an iterative method may require the use of a preconditioner if the system is ill-conditioned, none of the problems analyzed here are nearly ill-conditioned enough to merit the use of a preconditioner [15]. The cost required to build a preconditioner would not offset the performance benefit gained and therefore in our work we do not use any preconditioner.

Several centrality measures can be expressed as functions of the adjacency matrix of a graph [1]. Since powers of the adjacency matrix are used to count walks in networks, typically these centrality metrics weight vertices through some kind of walk counting. PageRank is a common method for ranking vertices in graphs, where a high score means random walks through the graph tend to visit the highly ranked vertices. PageRank can be thought of as either a global network centrality measure or a more personalized version where we only examine a local region of the large graph, and was first introduced rank webpages in a web search [16]. Given a search term from the user, PageRank incorporates a measure of a webpage's importance into the results of a set of webpages that could be relevant to the desired search term. However over time, many more applications have risen, such as in bibliometrics, social, and information network analysis. For example, personalized PageRank vectors have been used for local community detection [17]. It has also been used in analysis of road networks and for link prediction and recommendation systems [5]. To define the PageRank problem, we consider a random surfer model: a hypothetical random web surfer navigating between webpages online. When this random surfer visits a webpage, he tosses a coin; if the coin comes up heads he randomly clicks on a link from the current page and transitions there, if the coin comes up tails, he *teleports* to a (possibly random) page independent of the current page's identity.

Let $P = A^T D^{-1}$ be the transition matrix of probabilities, where D is the matrix of all diagonal values of A. Specifically $P(i, j)$ is the probability of transitioning from page j to page i. Assume the random surfer transitions according to the link structure of the web with probability α and teleports randomly with probability $1 - \alpha$. When teleporting randomly, the surfer teleports according to a teleportation distribution vector $\mathbf{v}$, where $\mathbf{v}$ is typically a uniform distribution over all pages. Many applications typically set α to 0.85 [5]. The solution $\mathbf{x}$ to Equation 2.1 gives the desired PageRank vector.

$$(I - \alpha P)\mathbf{x} = (1 - \alpha)\mathbf{v} \tag{2.1}$$

Eigenvector centrality is another linear algebra-based centrality measures for weighting relative importance of vertices in networks and does so by examining the eigenvector corresponding to the largest eigenvalue of the adjacency matrix [18]. Eigenvector centrality takes into account all walks through the network by considering both direct connections to vertices (edges to neighbors) as well as indirect (paths through the network). It is defined as the solution $\mathbf{x}$ to the equation $A\mathbf{x} = \lambda_{max}\mathbf{x}$, where λ_{max} is the largest eigenvalue of A, guaranteed to be positive and real by the Peron-Frobenius Theorem [19].

The subgraph centrality of a vertex weights walks in the graph of length k by a factor of $\frac{1}{k!}$ [20]. Recall that the number of walks of length k between vertices i and j is given by $[A^k](i, j)$. To calculate subgraph centrality scores we can derive the series $\sum_{k=0}^{\infty} A^k/k!$. The total subgraph communicability of a vertex is defined in terms of the row sums of matrix functions of the adjacency matrix of the network. The most common function is that of the matrix exponential in Equation 2.2.

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots + \frac{A^k}{k!} = \sum_{k=0}^{\infty} \frac{A^k}{k!} \tag{2.2}$$

The subgraph centrality of node i is given by $[e^A](i, i)$ while the subgraph communi-

cability between nodes i and j is given by $[e^A](i,j)$ [21]. A relatively high subgraph centrality indicates a more important vertex in the network and a high subgraph communicability between two vertices indicates that information flows more easily between those two vertices compared to other pairs of vertices with lower subgraph communicability.

A majority of the contributions in this dissertation address the ranking problem for Katz Centrality, a centrality metric that measures the affinity between vertices as a weighted sum of the walks between them [4]. Katz Centrality scores penalize long walks in the network through multiplication by a fixed, user-chosen factor α for each edge used, where $\alpha \in [0, \rho(A)]$. This gives rise to the series

$$\sum_{k=1}^{\infty} \alpha^{k-1} A^k,$$

where for $i \neq j$, the $(i,j)th$ element gives a weighted count of the number of walks of all lengths from vertex i to j. The ith sum of the series summarizes the ability of vertex i to initiate walks to all other vertices in the network and therefore the $(i,i)th$ element gives the weighted count of closed walks that start and finish at vertex i with a uniform shift. The Katz score of vertex i is therefore given as in Equation 2.3,

$$\mathbf{e}_i^T \sum_{k=1}^{\infty} \alpha^{k-1} A^k \mathbf{1}, \tag{2.3}$$

where $\mathbf{e}_i$ is the ith canonical basis vector, the vector of all 0s except a 1 in the ith position. In practice the Neumann formula [22] is employed to turn this series into a linear system and we compute the Katz Centrality ($\mathbf{c}$) of all vertices in the graph as in Equation 2.4.

$$\mathbf{c} = \sum_{k=1}^{\infty} \alpha^{k-1} A^k \mathbf{1} = A(I - \alpha A)^{-1} \mathbf{1} \tag{2.4}$$

2.3.2 Linear Algebra for Data Analysis Applications

Linear algebraic techniques as a tool for solving other data analysis problems has been studied in [23]. In fact, many data analysis problems are phrased as numerical problems for a more tractable solution, where the solution for the original data mining problem depends on the accuracy of the solution to the induced numerical problem. While there has been literature examining how the exact solution to the numerical problem affects the quality of the solution to the data mining problem, there has been little work in the realm of quantifying how the quality of the solution to the data mining problem is affected by an approximate solution to the numerical problem. In [24] this topic is addressed for spectral partioning. In spectral partitioning, vectors approximating eigenvectors of a graph matrix are used to partition a graph. The relationship between low-accuracy approximations of the eigenvectors is studied on the effect of the resulting partition. The authors conclude that although allowing more error in the eigenvalue computation potentially results in a loss of partition quality, the performance improvements in runtime are significant. In this dissertation, we address the topic of turning a data analysis problem into a numerical problem for centrality, where the data analysis problem is that of ranking vertices and the numerical problem is that of solving a linear system. Specifically we present theoretical results to quantify how accurate of a solution is needed when approximating a centrality vector using iterative solvers in order to accurately guarantee ranking of vertices in graphs.

For many application purposes it is primarily the highly-ranked vertices that are of interest. Consider performing a web search with Google. Anyone who runs a web search only cares about the top part of the ranking, or the most relevant results to the original search query; typically one only has enough human resources to examine the top. In a Twitter graph, we might wish to identify the most influential voices in a subset of Twitter users, or in a network modeling disease spread an analyst would be interested in finding sites of disease origin. Finally, consider a social network

modelling relationships between people. A common query is identifying the most active people in the network (or the most important). All these queries are answered by examining the highly ranked vertices in the respective graph.

As mentioned earlier, solving for many linear algebra based centrality measures directly is generally intractable so iterative solvers are used to approximate them [25]. By treating the ranking problem as obtaining a solution to a linear system, we present how error in the numerical approximation affects the solution to the original ranking problem. Understanding the error in the approximate solution to the numerical problem is key to understanding the error in the data mining problem. Ranking vertices in graphs and finding the top ranked vertices is of very practical relevance to data analysts. Relative importance of top vertices with respect to a particular seed set and ranking in practical settings are studied in [26].

We focus on approximating the centrality score of the vertices in the graph to a high enough accuracy to certify that the top of the ranking vector is accurate compared to the exact solution. Several other methods for approximating Katz scores across the network only examine walks up to a certain length [27] or employ low-rank approximation [28]. In [29], the authors provide theoretical guarantees for pairwise Katz scores. They use the Lanczos process to provide upper and lower bounds on the estimate of the pair-wise scores and exploit localization of the Katz matrix to provide estimates on the Katz scores. Our work differs in that we provide confidence as to which portion of the global ranking is correct and use the size of the residual to provide an accurate estimation of the ranking.

2.4 Dynamic Analysis of Centrality Measures

Since many real datasets are constantly evolving over time giving rise to a dynamic graph, much of today's graph analysis has focused on dynamic graph analysis. While much of the literature tends to focus on optimizing algorithms for centrality measures

on static graphs, a growing body of work addresses dynamic algorithms for updating centrality measures given updates to the underlying graph. Analytics that adapt quickly to changes in the graph are highly sought after, because otherwise re-compting them from scratch every time an update to the graph is made becomes very computationally expensive. In this section, we discuss literature on algorithms for centrality measures and community detection in dynamic graphs. Section 2.4.1 details literature about streaming algorithms for popular centrality measures that modify the algorithm itself, and Section 2.4.2 details literature for streaming centrality measures in a linear algebraic environment.

2.4.1 Popular centrality measures

Betweenness and closeness centrality are two very popular graph metrics in network analysis for identifying the most important vertices in a graph, with specific applications in network stability, traffic predictions, and social network analysis [1]. Betweenness centrality ($BC(v)$) looks at the vertices with high betweenness, i.e., those vertices whose removal would cause a significant number of shortest paths to not exist anymore. This notion was first established by Freeman, to compare the number of shortest paths going through a vertex v with the total number of shortest paths [30]. Formally the betweenness centrality score for vertex v is defined as $BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where $\sigma_{st}(v)$ is the number of shortest paths from vertex s to vertex t that include vertex v and σ_{st} is the number of shortest paths from s to t in general. Calculating the values can be done using Floyd-Warshall's all-pairs shortest-paths algorithm to find shortest paths from all vertices to all others in the graph. Applying Floyd-Warshall gives a runtime of $\mathcal{O}(|V|^3)$ [31], [32]. Johnson gave a faster method to calculate all-pairs shortest-paths in $\mathcal{O}(mn + n^2 \log n)$ [33]. Brandes provided a dependency accumulation technique to calculate betweenness centrality faster in $\mathcal{O}(|V||E|)$ [34], with an improved storage complexity. This technique is

faster for sparse networks; for dense networks, Floyd-Warshall's method is preferred.

Closeness centrality ($CC(v)$) was first introduced by Bavelas in 1950 to measure the 'farness' of a vertex, defined as the sum of its distances from all other vertices, and its 'closeness,' defined as the reciprocal of the farness [35]. Closeness centrality measures how close a vertex is to all other vertices based on the shortest-path length. The closeness centrality score for vertex v is defined as $CC(v) = \frac{1}{\sum_{t \in V} d_G(v,t)}$, where $d_G(v,t)$ is the length of the shortest path between vertices v and t. The exact closeness centrality value for each vertex in the graph can be calculated by solving the all-pairs shortest-paths problem in $\mathcal{O}(mn + n^2 \log n)$ [33], [36]. However, in large networks, calculating the exact value is often too computationally expensive and an approximation is sufficient. Eppstein provides an algorithm to approximate closeness centrality in $\mathcal{O}(\frac{\log n}{\epsilon^2}(n \log n + m))$, with an additive error of $\epsilon\Delta$ for the inverse of closeness centrality with probability $1 - \frac{1}{n}$ where $\epsilon > 0$ and Δ is the diameter of the graph [37]. Finally, if the values themselves are not important to the application, and only identification of vertices with high closeness centrality is required, a ranking method is of use. Okamoto developed a method to rank and obtain the top k vertices with the highest closeness centrality in the graph in $\mathcal{O}((k + n^{\frac{2}{3}} \cdot \log^{\frac{1}{3}} n)(n \log n + m))$ [38].

Since both these metrics are fairly computationally intensive to calculate, in the case of dynamic graphs it is optimal to have an algorithm that can update the centrality values with minimal effort as the graph updates instead of recomputing the centrality values from scratch. In [39], the authors propose an algorithm to update both betweenness and closeness calculations together after receiving edge updates to the graph. By splitting up the calculation of the centrality metrics into two parts, they avoid performing unneccessary calculations performed in previous timesteps. The first step repeats a calculation process until the shortest path is converged, and the second step aggregates the shortest path calculation into closeness and between-

ness centralities. The first step can be performed for both closeness and betweenness centrality simultaneously. The authors in [40] propose an incremental algorithm for closeness centrality by exploiting specific network topological properties: specifically their shortest-distance distributions, biconnected components distributions, and the existence of vertices with identical neighborhoods. They achieve a mean speedup of 43.5× for smaller graphs with less than 500K edges and 99.8× for larger graphs with more than 500K edges. Finally, the authors in [41] propose an incremental algorithm for updating betweenness centrality values by maintaining additional data structures to store previously computed values. They are able to achieve speedups of 100-400× on synthetic networks and speedups of 36-148× on real networks.

2.4.2 Incremental centrality using linear algebra

In this section we focus on dynamic algorithms for centrality measures based in a linear algebraic environment. As PageRank is one of the most commonly studied problems in the literature, we outline several dynamic algorithms for updating the centrality metric given edge updates to the graph. There are two general areas of techniques used to approximate dynamic updates to the PageRank vector: (1) linear algebraic methods that mainly use techniques from linear and matrix algebra and perhaps using some structural properties of the network [42, 43], and (2) Monte Carlo methods that use a small number of simulated random walks per vertex to approximate PageRank scores [44, 45]. Many linear algebraic techniques use "aggregation" methods, which operate under the assumption that changes to the underlying network affect only a localized portion of the PageRank vector [46, 47]. Aggregation methods partition the set of vertices into two disjoint sets S and $V \backslash S$, where S is the set of all vertices close to the incremental change and $V \backslash S$ is the set of all other vertices. All the vertices in $V \backslash S$ are aggregated into a single hyper-vertex and a smaller graph is created. The PageRank values of all the vertices are updated using this smaller graph

and the result is pushed back to the original graph. However, most aggregation techniques do not translate well for real-time applications due to both performance and quality reasons. Since the performance of these methods depends on the partitioning of the network, a poor partitioning can cause these methods to be extremely slow [48]. In terms of quality, since the aggregation is ultimately an approximation of the updated PageRank vector given incremental changes to the graph, the approximation error could potentially accumulate over time leading to a very poor quality PageRank vector. Monte Carlo methods for the incremental computation of approximate PageRank, personalized PageRank and similar random walk methods is examined in detail in [49]. These methods are typically very efficient and can achieve good quality personalized scores, but most literature on these approaches has thus far only been applied to static networks. These methods maintain a small number of short random walk segments starting at each vertex in the graph. For the case of identifying the top k vertices, these methods are able to provide highly accurate estimates of the centrality values for the top vertices, but smaller values in the personalized case are nearly identical and therefore impossible to tell apart. In [17], an algorithm for updating PageRank values in dynamic graphs by only using sparse updates to the residual is presented. In this thesis, we develop several algorithms to update Katz Centrality scores in dynamic graphs. To our knowledge, there is no method available to incrementally update Katz values in a dynamic graph without performing a full recomputation.

2.5 Community Detection

Community detection in graphs is a rapidly growing field of research and as such, there has been much work in the recent literature regarding development of algorithms for community detection. The task of community detection can broadly be thought of as identifying groups of vertices in a graph that are more closely related

to each other than the rest of the graph. For example, in a network modeling user behavior on Facebook, a community in that graph may be a group of friends who communicate most often with each other. Alternately, a community may be a set of people who belong and contribute to a specific page of interest on Facebook. Similarly, a community in a graph modeling financial transactions may be a group of individuals who primarily participate in mutual transactions with each other.

The definition of a community varies greatly amongst the existing literature. As such, there are several metrics that exist to evaluate the "quality" of a community. These metrics are further described in more detail in Chapter 5. Most of the literature in the field of community detection focuses on finding global communities in a static, unchanging graph. Popular methods include greedy agglomerative algorithms such as the Clauset-Newman-Moore (CNM) algorithm [50] and the Louvain method [51]. These two methods find a global partition of the graph in which each vertex belongs to exactly one community. Another widely used method of finding global, non-overlapping communities is spectral partitioning, which uses the eigenvectors of the Laplacian of the graph adjacency matrix to partition the graph in two [52, 53]. This process may be repeated recursively to find smaller communities.

While much of previous work has focused on partitioning methods that find non-overlapping communities, there are many methods that identify overlapping clusters. These include clique percolation [54], label propagation [55, 56], edge partitioning [57], Order Statistics Local Optimization Method (OSLOM) [58], multiple local expansions [59, 60, 61], and ensemble combinations [62].

Community detection has also been studied in the context of dynamic graph data and this work can be broadly divided into two categories. Algorithms in the first category focus only on quality, while those in the second aim to both detect high-quality communities and minimize computation. Typically, methods in the former category seek to find the best sequence of communities given the dynamic data by maximizing

both the quality of communities found at each point in time and the smoothness of community change over time. This can be done by collecting all temporal data before inferring communities, as in [63, 64, 65]. Using all temporal data may produce better choices of communities over time, but may be computationally expensive and can only be performed after all data is collected. Therefore, this approach is not suitable for applications in which updated communities must be found quickly. Alternatively, each community may be found using only past data, as in evolutionary clustering [66] and FaceNet [67].

In the other category of dynamic community detection work, the goal is to both maintain good communities on a changing graph while minimizing computation. Typically, this is done as follows. A graph is formed from an initial set of data and communities are identified. When the graph changes due to new data, new communities are found by starting with the previous community solution and incrementally updating it. Many algorithms of this type update the results of greedy, agglomerative algorithms. For example, Aynaud et al. [68] presents an incremental version of the Louvain algorithm. Whenever the graph changes, the previous clusters are used as a starting point. Changes are made by checking if the quality of the community would increase by moving any vertex to a different community. The work in [69] is another incremental version of Louvain clustering that starts with the previous cluster assignment modified based on the graph changes that occurred. The Modules Identification in Evolving Networks algorithm (MIEN) [70] is an incremental version of greedy agglomerative methods such as CNM. In the work by Riedy and Bader, whenever edges are inserted or deleted, the endpoint vertices of such edges are moved from their communities into singleton communities before restarting their agglomerative algorithm [71]. In the work in [72] by Görke et al., the authors present algorithms to maintain a clustering of a dynamic graph where edges appear as a stream by optimizing the quality while guaranteeing smoother clustering dynamics. Our work in the

field of community detection falls into this second category of dynamic community detection, except that we deal with local communities, which are described next.

Local community detection is the task of finding the best community for a set of vertices of interest, often called seed vertices. This is also called seed set expansion. When dealing with massive graphs, running computationally intensive analytics, visualization, and manual inspection by human analysts is likely to be infeasible, and this difficulty only increases for dynamic data. In such cases, local community detection can be used to extract a smaller, relevant subgraph in order to perform such tasks. Chapter 5 describes in more detail different local community detection methods that have previously been developed in the literature and how they relate to our work.

CHAPTER 3
NUMERICAL APPROXIMATIONS FOR CENTRALITY MEASURES

This chapter relates the two research areas of numerical analysis and data mining by turning the data analysis problem of ranking into a numerical problem of solving a linear system to some accuracy. We show that we can approximate the centrality scores of vertices on a graph to a high enough accuracy in order to guarantee vertex ranking in graphs. Here, we study Katz Centrality and PageRank. Theorems 1 and 2 present a new error bound on elements of a ranking vector to provide graph ranking guarantees to the computation of centrality. We turn our numerical theory into a new stopping criterion for iterative solvers in Section 3.1.2 to identify highly-ranked vertices in a graph that reduces runtime compared to running a solver to machine precision. We use Lanczos estimates to bound $\|A\|_2$, the matrix 2-norm of the adjacency matrix A in Section 3.1.3. Our analysis is applied to the computation of both global and personalized centrality scores and we develop sound theory with empirical analysis for both undirected and directed networks.

3.1 Theory

To solve for both Katz Centrality and PageRank, we are solving a linear system. When Katz Centrality was first introduced, Katz used the column sums of the matrix resolvent to obtain scores as $\mathbf{c}_{Katz} = A(I-\alpha A)^{-1}\mathbf{1}$ [4]. We refer to these as *global Katz scores*. From a graph perspective, these scores count the total number of weighted walks of all lengths ending at each vertex. We can also calculate *personalized Katz scores* from a particular vertex i, or more intuitively, weighted counts of the number of walks of all lengths starting at vertex i and ending at each vertex in the graph. These scores correspond to the ith column in the matrix $A(I-\alpha A)^{-1}$ and are calculated as

$\mathbf{c}_{Katz} = A(I - \alpha A)^{-1}\mathbf{e}_i$, where $\mathbf{e}_i$ is the ith canonical basis vector. Similarly, we can define personalized scores from a group of vertices $\mathcal{S} = \{v_1, v_2, \cdots, v_{|\mathcal{S}|}\}$ by defining a vector $\mathbf{e}_\mathcal{S} = \mathbf{e}_{v_1} + \mathbf{e}_{v_2} + \cdots + \mathbf{e}_{v_{|\mathcal{S}|}}$. The personalized scores w.r.t. $\mathcal{S}$ are then calculated as $\mathbf{c}_{Katz} = A(I - \alpha A)^{-1}\mathbf{e}_\mathcal{S}$. In this work when dealing with personalized scores we only use a single vertex, although the analyses presented can easily be extended to the group personalized case. The centrality scores obtained by Katz Centrality can thus be summarized as $\mathbf{c}_{Katz} = A\mathbf{x}_{Katz}$, where $\mathbf{x}_{Katz}$ is the solution to the linear system in Equation 3.1.

$$M_{Katz}\mathbf{x}_{Katz} = \mathbf{b}_{Katz} \tag{3.1}$$

We define $M_{Katz} = I - \alpha A$ and $\mathbf{b}_{Katz}$ to be either $\mathbf{1}$ or $\mathbf{e}_i$ depending on whether we are solving for the global or personalized Katz scores. Similarly, for PageRank we solve for the vector $\mathbf{c}_{PR} = (I - \alpha A^T D^{-1})^{-1}\mathbf{b}_{PR}$, or equivalently we solve the linear system $(I - \alpha A^T D^{-1})\mathbf{c}_{PR} = M_{PR}\mathbf{c}_{PR} = \mathbf{b}_{PR}$, where the right-hand side $\mathbf{b}_{PR}$ is set accordingly depending on whether we are solving for the global or personalized scores.

When the solution $\mathbf{c} = M^{-1}\mathbf{b}$ to either linear system is approximated, there will be differences between the approximate solution and the exact solution, where $\mathbf{c}$ is either $\mathbf{c}_{Katz}$ or $\mathbf{c}_{PR}$. We prove that these differences along with the ranking values can indicate how far down the ranking we can go before the approximation error makes it unreliable. For iteration k of the iterative solver, define $\mathbf{d}^{(k)} = \pi^{(k)}\mathbf{c}^{(k)}$, where $\pi^{(k)}$ is the permutation such that $\mathbf{d}^{(k)}$ is the vector $\mathbf{c}^{(k)}$ ordered in decreasing order so that $d_i^{(k)} \geq d_{i+1}^{(k)}$. Define $\lambda_{min}(M)$ to be the smallest eigenvalue of the matrix M and $\sigma_{min}(M)$ to be the smallest singular value of the matrix M, where M is either M_{Katz} or M_{PR}. Again recall that the residual norm is given as $r_k = \|\mathbf{b} - M\mathbf{x}^{(k)}\|_2$.

3.1.1 Error Analysis

We make the observation that if our goal is identification of the highly ranked vertices in a graph, we ought to focus on the *ranking accuracy* not *numerical accuracy*. This

is because the error in the data analysis problem of ranking is dfferent than the error in numerical problem of solving the linear system: the relative ranking of vertices can be correct even without a fully correct centrality vector. We theoretically guarantee the accuracy of the solution to numerical problem needed to successfully answer the data mining question of ranking for both Katz Centrality and PageRank.

Theorem 1 below provides guarantees as to when the rank of vertex i above j is correct from the approximate solution using Katz Centrality.

Theorem 1. *For undirected graphs, for any $i < j$, the rank of vertex i above j using Katz Centrality is correct if $|d_i^{(k)} - d_j^{(k)}| > 2\epsilon_k$ for $\epsilon_k = \frac{\|A\|_2}{\lambda_{min}(M_{Katz})} r_k$. For directed graphs, for any $i < j$, the rank of vertex i above j is correct if $|d_i^{(k)} - d_j^{(k)}| > 2\widetilde{\epsilon}_k$ for $\widetilde{\epsilon}_k = \frac{\|A\|_2}{\sigma_{min}(M_{Katz})} r_k$.*

Proof. Using foundations of error analysis in linear solvers, we can bound the componentwise error in the ranking, which will then provide a sufficient error gap in the elements of the approximation to the ranking vector.

$$\begin{aligned}
\|\mathbf{d}_{Katz}^* - \mathbf{d}_{Katz}^{(k)}\|_\infty &= \|\mathbf{c}_{Katz}^* - \mathbf{c}_{Katz}^{(k)}\|_\infty \\
&\leq \|\mathbf{c}_{Katz}^* - \mathbf{c}_{Katz}^{(k)}\|_2 \\
&= \|A\mathbf{x}_{Katz}^* - A\mathbf{x}_{Katz}^{(k)}\|_2 \\
&\leq \|A\|_2 \|\mathbf{x}_{Katz}^* - \mathbf{x}_{Katz}^{(k)}\|_2 \\
&= \|A\|_2 \|M_{Katz}^{-1}\mathbf{b}_{Katz} - \mathbf{x}_{Katz}^{(k)}\|_2 \\
&\leq \|A\|_2 \|M_{Katz}^{-1}\|_2 \|\mathbf{b}_{Katz} - M_{Katz}\mathbf{x}_{Katz}^{(k)}\|_2 \\
&\leq \|A\|_2 \|M_{Katz}^{-1}\|_2 r_k
\end{aligned}$$

For undirected graphs (with A symmetric), we have $\|M_{Katz}\|^{-1} \leq \frac{1}{\lambda_{min}(M_{Katz})}$, so we

can write:

$$\|\mathbf{d}^*_{Katz} - \mathbf{d}^{(k)}_{Katz}\|_\infty \leq \frac{\|A\|_2}{\lambda_{min}(M_{Katz})} r_k \tag{3.2}$$
$$=: \epsilon_k$$

For directed graphs (with A nonsymmetric), $\|M_{Katz}\|^{-1}$ is bounded by the inverse of the minimum singular value instead of the inverse of the minimum eigenvalue:

$$\|\mathbf{d}^*_{Katz} - \mathbf{d}^{(k)}_{Katz}\|_\infty \leq \frac{\|A\|_2}{\sigma_{min}(M_{Katz})} r_k \tag{3.3}$$
$$=: \widetilde{\epsilon}_k$$

Let $d(i)$ be the value of the ith vertex in the graph. Since $d(i)^{(k)}_{Katz} - d(i)^*_{Katz} < \epsilon_k$ and $d(j)^*_{Katz} - d(j)^{(k)}_{Katz} < \epsilon_k$, this means that $d(i)^*_{Katz} - d(j)^*_{Katz} > d(i)^{(k)}_{Katz} - d(j)^{(k)}_{Katz} - 2\epsilon_k$. If $d(i)^{(k)}_{Katz} - d(j)^{(k)}_{Katz} > 2\epsilon_k$, then $d(i)^*_{Katz} - d(j)^*_{Katz} > 0$ meaning that the ranking of vertex i above j is correct. □

Similarly, we can derive a corresponding bound for PageRank to guarantee the ranking of vertices in the approximate ranking vector. We again separate the bounds into the undirected and directed graph cases.

Theorem 2. *For undirected graphs, for any $i < j$, the rank of vertex i above j is correct using PageRank if $|d^{(k)}_i - d^{(k)}_j| > 2\epsilon_k$ for $\epsilon_k = \frac{1}{\lambda_{min}(M_{PR})} r_k$. For directed graphs, for any $i < j$, the rank of vertex i above j is correct using PageRank if $|d^{(k)}_i - d^{(k)}_j| > 2\widetilde{\epsilon}_k$ for $\widetilde{\epsilon}_k = \frac{1}{\sigma_{min}(M_{PR})} r_k$.*

Proof.

$$\begin{aligned}\|\mathbf{d}^*_{PR} - \mathbf{d}^{(k)}_{PR}\|_\infty &= \|\mathbf{c}^*_{PR} - \mathbf{c}^{(k)}_{PR}\|_\infty \\ &\leq \|\mathbf{c}^*_{PR} - \mathbf{c}^{(k)}_{PR}\|_2 \\ &= \|M^{-1}_{PR}\mathbf{b}_{PR} - \mathbf{x}^{(k)}_{PR}\|_2 \\ &\leq \|M^{-1}_{PR}\|_2\|\mathbf{b}_{PR} - M_{PR}\mathbf{x}^{(k)}_{PR}\|_2 \\ &\leq \|M^{-1}_{PR}\|_2 r_k\end{aligned}$$

For undirected graphs (with A symmetric), we have $\|M_{PR}\|^{-1} \leq \frac{1}{\lambda_{min}(M_{PR})}$, so we can write:

$$\begin{aligned}\|\mathbf{d}^*_{PR} - \mathbf{d}^{(k)}_{PR}\|_\infty &\leq \frac{1}{\lambda_{min}(M_{PR})} r_k \\ &=: \epsilon_k\end{aligned} \tag{3.4}$$

For directed graphs (with A nonsymmetric), $\|M_{PR}\|^{-1}$ is bounded by the inverse of the minimum singular value instead of the inverse of the minimum eigenvalue:

$$\begin{aligned}\|\mathbf{d}^*_{PR} - \mathbf{d}^{(k)}_{PR}\|_\infty &\leq \frac{1}{\sigma_{min}(M_{PR})} r_k \\ &=: \widetilde{\epsilon}_k\end{aligned} \tag{3.5}$$

Again, since $d(i)^{(k)}_{PR} - d(i)^*_{PR} < \epsilon_k$ and $d(j)^*_{PR} - d(j)^{(k)}_{PR} < \epsilon_k$, this means that $d(i)^*_{PR} - d(j)^*_{PR} > d(i)^{(k)}_{PR} - d(j)^{(k)}_{PR} - 2\epsilon_k$. If $d(i)^{(k)}_{PR} - d(j)^{(k)}_{PR} > 2\epsilon_k$, then $d(i)^*_{PR} - d(j)^*_{PR} > 0$ meaning that the ranking of vertex i above j is correct. □

We observe in practice that the bounds in Theorems 1 and 2 are tight enough to produce relevant results in many practical applications (seen in Section 3.2) and lend themselves to the development of a new stopping criterion for iterative solvers when identifying the highly ranked vertices in a graph.

3.1.2 New Stopping Criterion

Current methods for identifying the top vertices in a graph involve running an iterative solver to machine precision to obtain an approximation of $\mathbf{c}^*$. We introduce a new stopping criterion to find these top vertices that typically provides results much faster than existing methods, based off of the theory developed in Theorems 1 and 2 above. Furthermore, our method provides theoretically sound guarantees as to the correctness of the top vertices, unlike the common method of simply running a solver to machine precision and blindly hoping the resulting vector is good enough for the desired data mining task.

Suppose a user desires a set of j vertices containing the top R highly ranked vertices in a graph, with precision ϕ^*. How large does j need to be before we can accurately certify (or guarantee) that the top vertices are in the set? We are not concerned with the internal ordering of this set, but rather that the top R vertices are contained somewhere within the superset of j vertices. The desired precision ϕ^* gives a sense of how many false positives we will tolerate in our set. We answer this question using our theory.

Here, we present the implementation for the theory for Katz Centrality on undirected graphs, but the same principle can be applied to develop a stopping criterion for PageRank or directed networks. For brevity, we drop the $Katz$ subscript in this section. This procedure is given in Algorithm 4, for an adjacency matrix A, right-hand side $\mathbf{b}$, number of top vertices R, desired precision ϕ^*, maximum number of iterations k_{max}, and upper bound σ_{up} on $\|A\|_2$. Note we discuss bounds for $\|A\|_2$ in the next section. At each iteration of conjugate gradient, the current solution $\mathbf{c}^{(k)}$ is ordered in decreasing order to produce the vector $\mathbf{d}^{(k)}$ as described earlier. We find the first position $j > R$ in $\mathbf{d}^{(k)}$ where we find the necessary gap of $|d_R^{(k)} - d_j^{(k)}| > 2\epsilon_k$. The precision for these values of R and j is defined as $\phi = \frac{R}{j-1}$. If for this value of j we have the desired precision ϕ^*, meaning $\phi = \frac{R}{j-1} \geq \phi^*$, then we terminate, else we

iterate again using conjugate gradient to obtain a more accurate approximation.

Intuitively the precision shows how far past position R we must travel down the vector to find the necessary gap to ensure we are returning the top R vertices in the graph. Conjugate gradient can be organized to return $\mathbf{x}^{(k)}$, $\mathbf{c}^{(k)}$, and the residual norm r_k at each iteration (denoted CGITERATION in Algorithm 4).

Algorithm 4 Obtain top R vertices in network with precision ϕ^*

1: **procedure** TOP_R($A, \mathbf{b}, R, \phi^*, k_{max}, \sigma_{up}$)
2: $k = 0$; $j = \infty$
3: $M = I - \alpha A$
4: **while** $\frac{R}{j-1} < \phi^*$ and $k < k_{max}$ **do**
5: $\mathbf{x}^{(k)}, \mathbf{c}^{(k)}, r_k =$ CGITERATION$(M, \mathbf{x}^{(k-1)}, \mathbf{b})$
6: $\mathbf{d}^{(k)} = \pi^{(k)}\mathbf{c}^{(k)}$ ▷ Sort $\mathbf{c}^{(k)}$ in descending order
7: $\epsilon_k = \frac{\sigma_{up}}{\lambda_{min}(M)} r_k$
8: $j = argmin_{i>R} |d_R^{(k)} - d_i^{(k)}| > 2\epsilon_k$
9: $k \mathrel{+}= 1$

To solve for PageRank instead of Katz Centrality, we modify Line 2 to $M = I - \alpha A^T D^{-1}$ and change the bound accordingly in Line 4. For the directed graph case, we use GMRES instead of CONJUGATE_GRADIENT in Line 2 and again modify the bound in Line 4. The vector $\mathbf{b}$ is set to $\mathbf{1}$ or $\mathbf{e}_i$ accordingly depending on if we are solving for the global or personalized scores. The procedures for conjugate gradient or GMRES are given previously in Algorithms 2 or 3 respectively.

3.1.3 Bounds on $\|A\|_2$

We obtain a tight bound on ϵ_k which allows us to certify that the ranking of vertex i above j is correct if the gap between two elements in the ranking vector is greater than our error bound, $|d_i^{(k)} - d_j^{(k)}| > 2\epsilon_k$. The iterative solver can be organized to readily provides the residual norm r_k at each iteration, and $\lambda_{min}(M)$ or $\sigma_{min}(M)$ can be computed provided α is chosen in the given range. To certify portions of the ranking vector, we desire ϵ_k to be as small as possible to find places in the vector where the necessary gap $|d_i^{(k)} - d_j^{(k)}|$ exists. For the bounds on Katz Centrality, obtaining a

tight bound on $\|A\|_2$ is key to bounding ϵ_k; we present and compare two methods of bounding $\|A\|_2$.

The Gershgorin Circle Theorem [73] bounds the eigenvalues of the symmetric matrix A. Let $T_i = \sum_{j \neq i} |a_{ij}|$, the sum of the nondiagonal entries in row i. Then $D(a_{ii}, T_i)$ is the closed interval centered at a_{ii} with radius T_i and every eigenvalue $\lambda \in \sigma(A)$ must lie within at least one interval $D(a_{ii}, T_i)$, where $\sigma(A)$ is the spectrum of A. Since the diagonal entries a_{ii} of A are 0, the discs are all centered around the origin and $\forall i, T_i = d_i =$ the degree of vertex i. We then have $\|A\|_2 = \max \lambda_i < \max T_i = d_{max}$, where d_{max} is the largest degree in the graph. While this provides a basis for an upper bound of the matrix 2-norm of A, many real-world graphs such as social networks have a scale-free distribution and thus contain vertices with a very large degree [74]. Therefore, this is often a non-optimal bound. By using just a few matrix-vector multiplications applied to random vectors, we can compute tighter bounds with high certainty.

We next examine probabilistic matrix norm bounds [75] and consider replacing the true bound σ_{up} with an estimate of a bound with some probability. These bounds are developed using the polynomials p, q implicitly formed as a part of the Lanczos bidiagonalization process with starting vector $\mathbf{v_1}$, which is chosen randomly with unit norm. For $\beta_0 = 0$ and $\mathbf{u}^{(0)} = 0$ and $k \geq 1$, the defining relations of Lanczos bidiagonalization are stated as

$$\gamma_j \mathbf{u}^{(j)} = A\mathbf{v}^{(j)} - \beta_{j-1}\mathbf{u}^{(j-1)}$$
$$\beta_j \mathbf{v}^{(j+1)} = A^T\mathbf{u}^{(j)} - \gamma_j \mathbf{v}^{(j)},$$

where $\gamma_j = {\mathbf{u}^{(j)}}^T A\mathbf{v}^{(j)}$ and $\beta_j = {\mathbf{u}^{(j)}}^T A\mathbf{v}^{(j+1)}$ are nonnegative. Therefore the following

recurrence relations hold for the recurrent polynomials derived as below:

$$\gamma_{j+1}p_j(t) = q_j(t) - \beta_j p_{j-1}(t)$$
$$\beta_{j+1}q_{j+1}(t) = tp_j(t) - \gamma_{j+1}q_j(t),$$

for $p_{-1}(t) = 0$ and $q_0(t) = 1$ for $j \geq 0$. The bound is stated in Theorem 3 and the algorithm from [75] is reproduced here for clarity. Note in Algorithm 5 that the matrices U and V are the concatenated column vectors $\mathbf{u}_j$ and $\mathbf{v}_j$ respectively. The result is an upper bound $\sigma_{up}(\theta)$ for $\|A\|_2$ with probability 1-θ, where θ is the user-chosen probability of bound failure. Define $\delta = \theta \cdot \frac{1}{2}B(\frac{n-1}{2}, \frac{1}{2})$ where B is Euler's Beta function, $B(x, y) = \int_0^1 t^{x-1}(1-t)^{y-1}dt$.

Theorem 3. *[75] Suppose we have carried out k steps of the Lanczos bidiagonalization process with starting vector* $\mathbf{v_1}$*, and let* $\theta \in (0, 1)$*. Then the largest zero of the polynomials,*

$$f_1(t) = q_k(t^2) - 1/\delta, f_2(t) = tp_k(t^2) - 1/\delta$$

with δ *given above, is an upper bound* $\sigma_{up}(\theta)$ *for* $\|A\|_2$ *with probability at least 1-*θ*.*

As a result of thorough experimentation, for all bounds used in this section, we select values of θ=0.01 and k=10. For k=10, in order to calculate $\sigma_{up}(0.01)$ we are required to calculate the largest root of a tenth order polynomial. Since this does not change regardless of problem size n, this calculation is asymptotically a fixed cost. We use Python's SYMPY package to calculate the roots of these polynomials. The deterministic Gershgorin bounds yield large values of $\|A\|_2$, rendering these bounds useless. On average, these bounds return estimates of $\|A\|_2$ that are 30.9× greater than the true 2-norm. In contrast, the probabilistic bounds presented in Theorem 3 return estimates of $\|A\|_2$ that are only on average 1.07× greater than the true 2-norm,

Algorithm 5 Lanczos bidiagonalization to calculate probabilistic upper bound $\sigma_{up}(\theta)$ on $\|A\|_2$ with probability θ

procedure CALC_UPPER_BOUND(A, $\mathbf{v}^{(1)}$, θ)
  $\delta = \theta \cdot \frac{1}{2}B(\frac{n-1}{2}, \frac{1}{2})$
  $p_{-1}(t) = 0$, $q_0(t) = 1$
  for j=1$\cdots k$ **do**
    $\mathbf{u} = A\mathbf{v}^{(j)}$
    if j >1 **then**
      $\mathbf{u} = \mathbf{u} - \beta_{j-1}\mathbf{u}^{(j-1)}$
      $\mathbf{u} = \mathbf{u} - U_{j-1}(\mathbf{u}^T U_{j-1})^T$
    $\gamma_j = \|\mathbf{u}\|$
    $\mathbf{u}_j = \mathbf{u}/\gamma_j$
    $\mathbf{v} = A^T\mathbf{u}$
    $\mathbf{v} = \mathbf{v} - \gamma_j\mathbf{v}^{(j)}$
    $\mathbf{v} = \mathbf{v} - V_j(\mathbf{v}^T V_j)^T$
    $\beta_j = \|\mathbf{v}\|$
    $\mathbf{v}^{(j+1)} = \mathbf{v}/\beta_j$
    $p_j(t) = \frac{q_j(t) - \beta_j p_{j-1}(t)}{\gamma_{j+1}}$
    $q_{j+1}(t) = \frac{tp_j(t) - \gamma_{j+1}q_j(t)}{\beta_{j+1}}$

meaning that these are able to be used for practical purposes.

Remark 1. Future work will examine obtaining the bound in real-time without any additional computational cost. In the Lanczos algorithm to obtain σ_{up} we are applying A to obtain $\mathbf{u} = A\mathbf{v}$, and in conjugate gradient we are applying A to obtain $(I - \alpha A)\mathbf{x}^{(k)}$ in each iteration. These two operations can be combined and we can apply A to both vectors in the same algorithm, effectively performing both Algorithms 4 and 5 simultaneously, which is important for distributed implementations of these algorithms.

3.2 Results

In this section we present comparisons to existing methods for identifying the top ranked vertices with respect to performance and experiments validating our bound with respect to precision. We are interested in determining if our method correctly

Table 3.1: Undirected graphs used in numerical experiments. Columns are graph name, number of vertices, number of edges, and type of graph.

Graph	$\|\boldsymbol{V}\|$	$\|\boldsymbol{E}\|$	**Type**
douban	154,908	327,162	social
gowalla	196,591	950,327	social
dblp	317,080	1,049,866	coauthorship
dogster	426,820	8,546,581	social
catster	623,766	15,699,276	social
youtube	1,134,890	2,987,624	social
skitter	1,696,415	11,095,298	computer
flickr	1,715,255	15,551,250	social
california	1,965,206	2,766,607	infrastructure
facebook	63,731	817,035	social
pgp	10,680	24,316	online
livejournal	5,204,175	49,174,464	social
orkut	3,072,441	117,184,899	social

Table 3.2: Directed graphs used in numerical experiments. Columns are graph name, number of vertices, number of edges, and type of graph.

Graph	$\|\boldsymbol{V}\|$	$\|\boldsymbol{E}\|$	**Type**
edinburgh	23,132	312,342	lexical
cora	23,166	91,500	citation
lkml	63,399	1,096,440	social
epinions	75,879	508,837	social
enron	87,273	1,148,072	social
baidu	2,141,300	17,794,839	hyperlink
wiki-german	3,225,565	8,1626,917	hyperlink
wiki-english	18,268,991	172,183,984	hyperlink

identifies the set of top vertices and if so, how much faster we are able to certify this set. The common method of iterating to machine precision does not theoretically certify this set but our theory can be used on the machine precision solution as well. We conduct experiments on both undirected and directed networks from the KONECT [76] collection, including social networks, autonomous systems, citation, co-authorship, and web graphs. Table 3.1 gives the undirected networks used and Table 3.2 gives the directed networks used.

For the results shown here, we vary values of the desired precision as

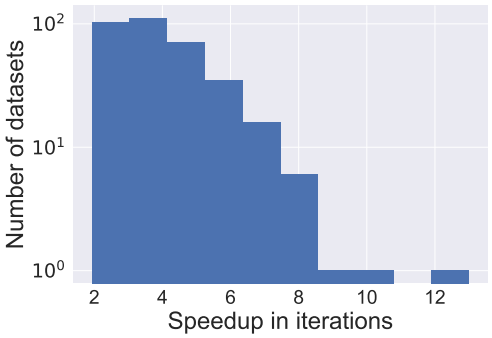
$\phi \in \{0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ and the top R as $R = 10$, 100, and 1000. For Katz Centrality, we vary the α parameter as a fraction of its upper bound $1/\|A\|_2$. For personalized centrality results, we form the vector $\mathbf{e}_i$ by choosing a vertex i randomly from the top 10% of highest degree vertices.

3.2.1 Speedup in iterations

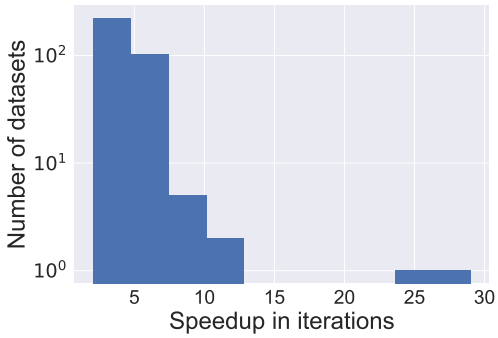
We first analyze the effect of our stopping criterion on reducing the number of iterations taken by an iterative solver to identify the top R vertices in a graph. We denote the number of iterations taken by either conjugate gradient/GMRES to converge to machine precision as I_E and the number of iterations using our new stopping criterion as I_A and calculate speedup w.r.t. number of iterations as

$$speedup = \frac{I_E}{I_A}.$$

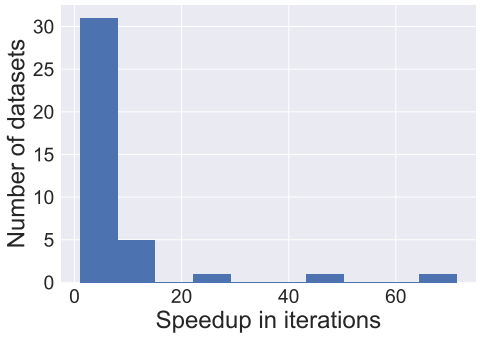
In this section we only show results obtained with a precision of 1.0 (so for a desired set of the top R vertices we return a set guaranteed to have no false positives) and we show results for all values of R (10, 100, and 1000). For Katz Centrality results, we sample all values of α as well. Figure 3.1 plots the distribution of the speedups for undirected graphs. Figures 3.1a and 3.1b plot the histograms for global and personalized Katz Centrality scores, respectively, and Figures 3.1c and 3.1d show global and personalized results for PageRank, respectively. For the undirected graphs, for Katz scores we have an average of 3.99× speedup for global scores and 4.03× for personalized scores, and for PageRank an average of 6.24× speedup for global scores and 10.23× for personalized scores. Figure 3.2 plots the distribution of the speedups for directed graphs, again for global and personalized Katz and PageRank scores. For the directed networks, for Katz scores we obtain an average of 4.60× speedup for global scores and 5.04× for personalized scores, and for PageRank an average



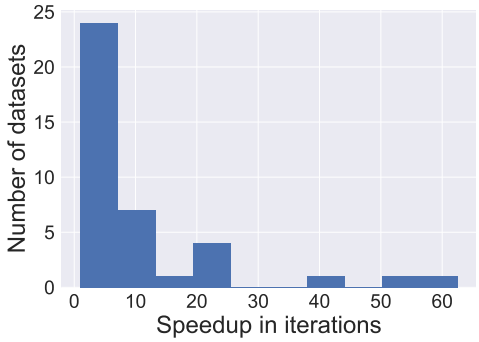
(a) Speedup for global Katz scores.



(b) Speedup for personalized Katz scores.



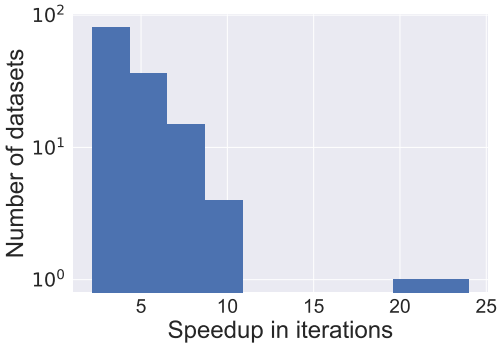
(c) Speedup for global PageRank scores.



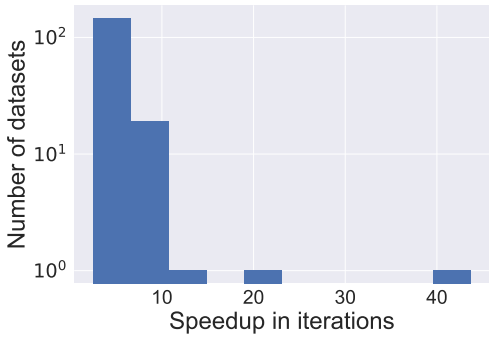
(d) Speedup for personalized PageRank scores.

Figure 3.1: Histograms of speedups in iterations for undirected graphs with precision 1.0. Higher values of speedup are better.

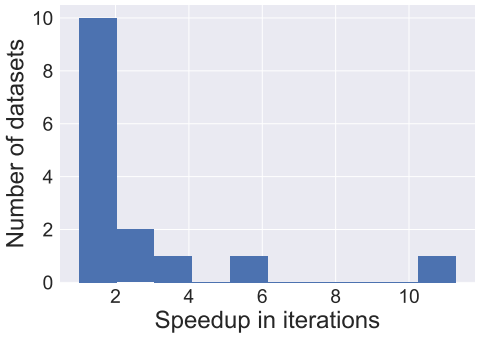
of 2.52× speedup for global scores and 23.91× for personalized scores. In all cases we obtain a speedup greater than 1× and up to a speedup of a maximum of over two orders of magnitude. This shows that we are able to identify the top R in a fraction of the time using our stopping criterion compared to running until machine precision, while providing a theoretical guarantee that these vertices are in the top of the ranking vector. This is especially significant because running to machine precision can sometimes take hundreds or thousands of iterations.



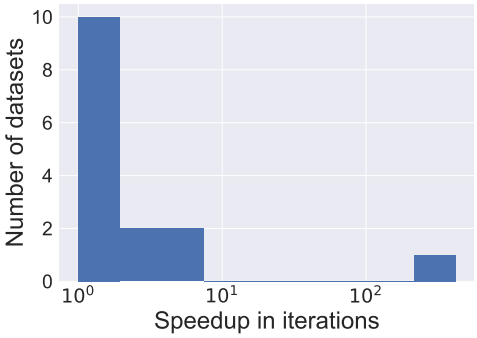
(a) Speedup for global Katz scores.



(b) Speedup for personalized Katz scores.



(c) Speedup for global PageRank scores.



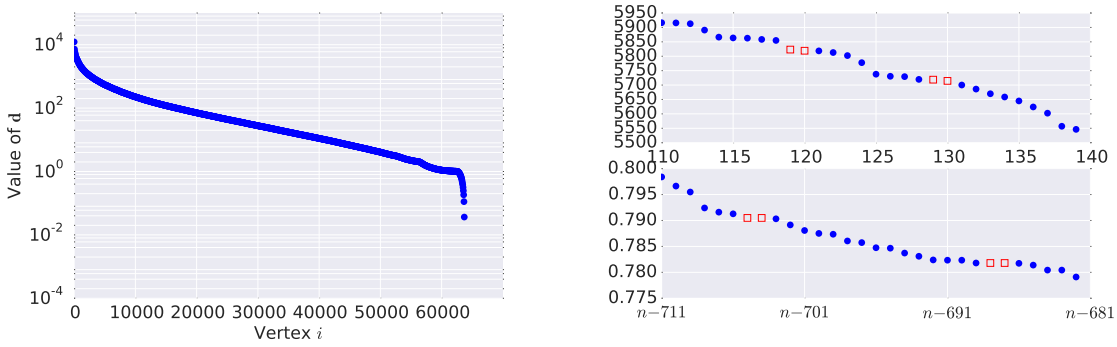
(d) Speedup for personalized PageRank scores.

Figure 3.2: Histograms of speedups in iterations for directed graphs with precision 1.0. Higher values of speedup are better.

3.2.2 Performance vs. quality

We have shown that we are able to obtain speedups w.r.t. iteration counts using our theory versus running an iterative solver to machine precision. In this section we examine the effect varying the precision of the returned set of top vertices has on the speedup obtained.

We first explain the behavior of the sorted ranking vector $\mathbf{d}$ of a single undirected graph, *facebook*, a citation network, using Katz Centrality in Figure 3.3. Figure 3.3a plots the sorted values of $\mathbf{d}$ on a log-scale for all the vertices and Figure 3.3b zooms in on selected regions from Figure 3.3a. The top plot of Figure 3.3b shows values for vertices 110-140 (vertices at the beginning of the sorted vector) and the bottom plot



(a) All vertices.

(b) Specific subsets of vertices.

Figure 3.3: Sorted ranking vector $\mathbf{d}_{Katz}$ for *facebook* graph. Values are plotted in blue circles while selected points with an extremely close error gap are shown in red squares. Left plot is on a log-scale; right plots are on a linear scale.

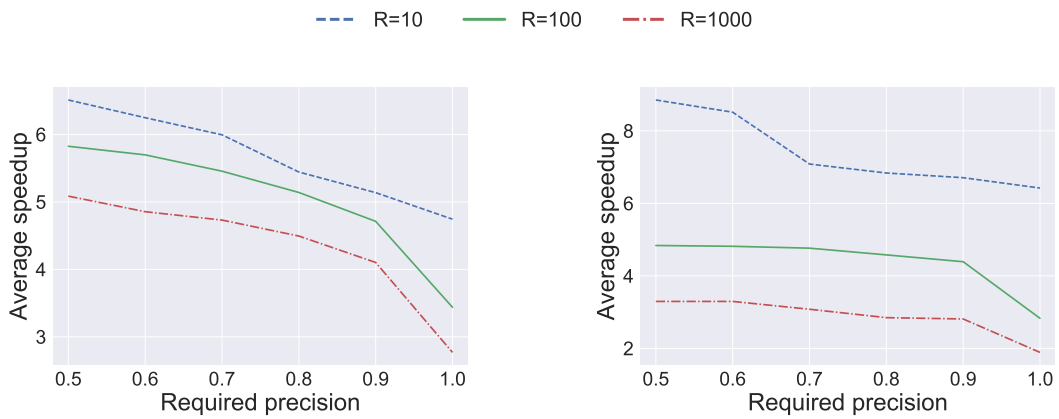
shows values for vertices $n - 711$-$n - 681$ (vertices with scores toward the end of the vector). The value of ϵ_k obtained as a part of our theory is absolute. We are able to resolve the part of the vector that the data mining task cares about, namely the top of the vector (the highly ranked vertices), with a guarantee that they are correct compared to the exact solution. However, for another use case where the user desires all the vertices in the graph to be returned correctly, since the values typically get closer to each other the further one traverses down the ranking vector, the value of ϵ_k will not be sufficient to provide the necessary gap between two elements toward the end of the vector. This is seen in Figure 3.3b. For the top right plot, the two pairs of open red squares indicate pairs of vertices where the gap is sufficient to certify the ranking of one vertex above the other. Using our previous notation, this is translated into a required precision of 1.0 (where we look for gaps between sucessive vertices). For the first pair, the difference in the scores is $9.4 \times 10^9 \times 2\epsilon_k$ and the difference between the second pair of vertices is $9.9 \times 10^9 \times 2\epsilon_k$. However, in the bottom right plot (values for vertices further down the ranking vector) where the values are very close together, the required gap $2\epsilon_k$ is larger than the difference between successive pairs of points. The two pairs of open red squares indicate pairs of vertices with

values too close together to obtain the necessary gap.

Overall the $\mathbf{d}$ vector follows an exponential decay pattern. The plateau-like behavior of the vector at certain points that is more clearly seen in Figure 3.3b can be explained by the fact that the Katz vector tends to have sets of vertices grouped so tightly together around the same value that we are unable to have the necessary separation to apply the error analysis to certify individual vertices' ranking. Therefore, when we want the top R vertices, it is sometimes necessary to travel further down the ranking vector to $j = R + \Delta$ to obtain the required separation between vertices, where Δ is the number of false positives returned in the set, or equivalently, obtain highly ranked vertices with less than perfect (1.0) precision.

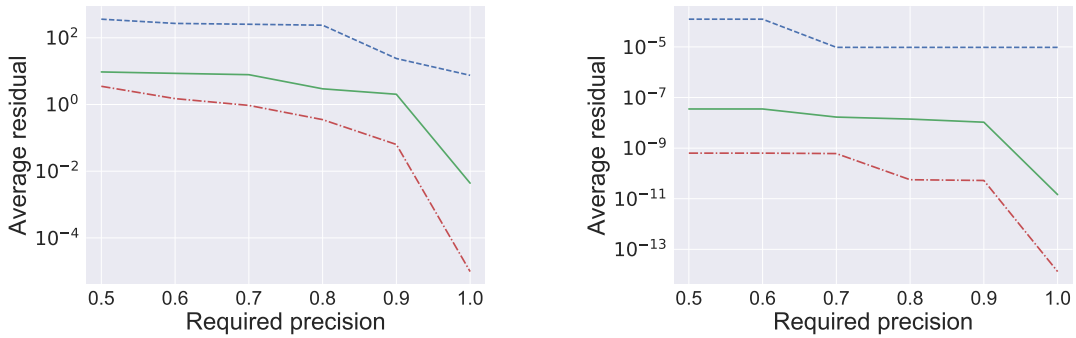
Next we examine the tradeoff between performance and quality of our algorithm. Recall for the top R vertices returned in a superset of j vertices, we define precision as $\frac{R}{j-1}$. Requiring a predetermined precision of ϕ^* means we want $\frac{R}{j-1} > \phi^*$. Figure 3.4 plots the average speedup and terminating residual for global and personalized scores for Katz Centrality on undirected graphs, where the terminating residual is the residual upon terminating at our new stopping criterion (iteration $k = I_A$). We plot results for $\alpha = \frac{0.9}{\|A\|_2}$, although trends seen for other values of α are similar. Figures 3.4a and 3.4b plot the average speedup versus required precision in iterations for global and personalized scores respectively, and Figures 3.4c and 3.4d plot the terminating residual versus required precision for global and personalized scores respectively. All plots show results for the top $R = 10$, 100, and 1000 vertices.

In all cases (for both speedup and terminating residual), we have more of an improvement using our stopping criterion for smaller values of R. More specifically, we obtain greater speedups and are able to terminate at a higher residual (obtaining a less accurate numerical solution) for smaller values of R. This behavior can be attributed to the shape of the centrality vector as explained by Figure 3.3 previously. While the gap $2\epsilon_k$ that we are looking for in between elements of the centrality vec-



(a) Speedup versus precision for global Katz scores.

(b) Speedup versus precision for personalized Katz scores.



(c) Terminating residual versus precision for global Katz scores.

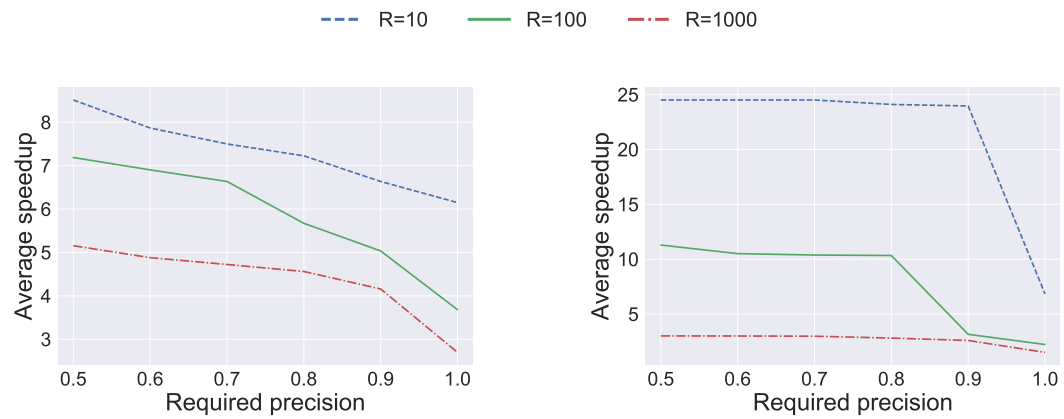
(d) Terminating residual versus precision for personalized Katz scores.

Figure 3.4: Performance versus required precision for Katz Centrality on undirected graphs (with $\alpha = 0.9/\|A\|_2$).

tor is fixed, elements in the vector themselves decrease exponentially. Therefore, for larger values of R we need to traverse further down the ranking vector to obtain the necessary gap. Nevertheless, we still see significant speedups for larger values of R such as 1000. In all cases, even for large R and high precision rates, we are able to terminate at a residual significantly above machine precision. For the personalized results (Figures 3.4b and 3.4d), we see a greater speedup but lower terminating residual than their global counterparts (Figures 3.4a and 3.4c). Intuitively, we obtain smaller terminating residuals for the personalized results because the values in the ranking vector themselves are smaller. For a possible reason behind the greater speedup in

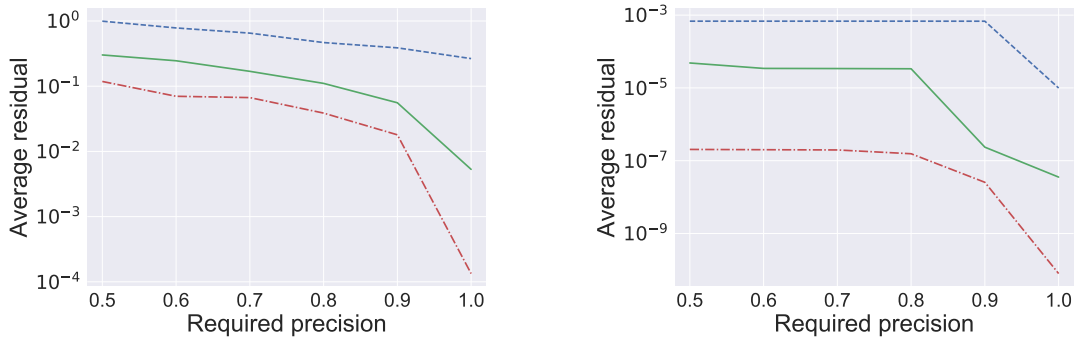
the personalized case, we turn our attention back to the theory presented in Theorem 1. Our stopping criterion terminates the iterative solver when we have a necessary gap between elements in the ranking vector of $2\epsilon_k = 2\frac{\|A\|_2}{\lambda_{min}} r_k$, where r_k is the residual norm. The gap ϵ_k differ in the global and personalized case only in the residual norm. Therefore, the residual dictates how far we need to traverse down the ranking vector until we can guarantee the top vertices in the returned set. Since the residual in the personalized case is several orders of magnitude smaller than the residual in the global case, we seek a smaller gap between elements in the ranking vector. We are therefore able to stop after fewer iterations, relative to machine precision, in the personalized case. Finally as expected, as we increase the required precision we see reduced speedups and smaller terminating residuals. Increasing the required precision means we desire a tighter set of the top R vertices to be returned. For example, for a precision of 1.0 we are looking for a gap of $2\epsilon_k$ between elements R and $R+1$, whereas for a precision of 0.5 we are only looking for a gap between elements R and $2R+1$. Clearly we will be able to find a gap between elements that are farther apart such as R and $2R+1$ much faster than successive elements R and $R+1$, so larger speedups for smaller precisions is not surprising. However, we note that the difference in speedups for required precisions from about 0.5 to 0.9 is about the same as the difference in speedups for required precisions from about 0.9 to 1.0. This means that we are able to quickly obtain highly ranked vertices without sacrificing too much quality.

Figure 3.5 broadly plots the same results as above except for directed graphs. We again plot results for $\alpha = \frac{0.9}{\|A\|_2}$. Figures 3.5a and 3.5b plot the average speedup versus required precision in iterations for global and personalized scores respectively, and Figures 3.5c and 3.5d plot the terminating residual versus required precision for global and personalized scores respectively. Most of the same trends discussed from the undirected results are applicable for the directed graphs. In fact, for the personalized speedups (Figure 3.4b), there is a much stronger trend of obtaining a



(a) Speedup versus precision for global Katz scores.

(b) Speedup versus precision for personalized Katz scores.



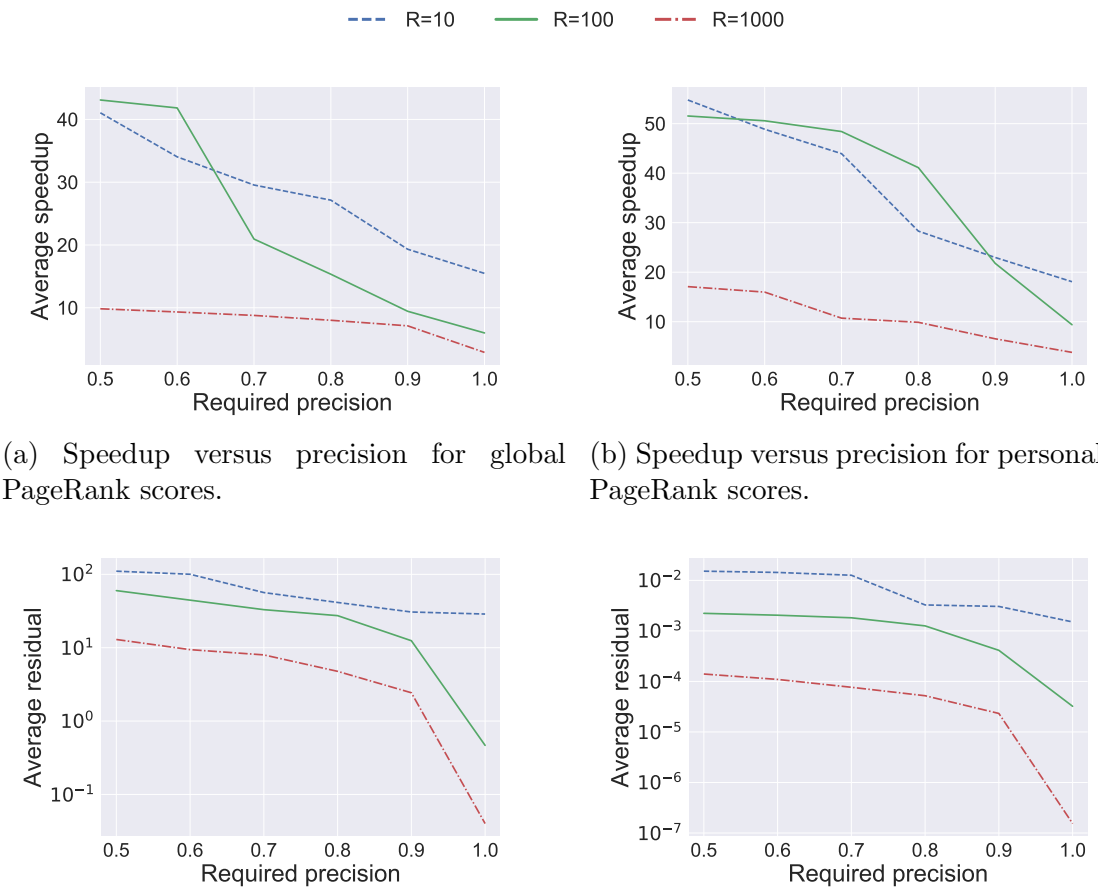
(c) Terminating residual versus precision for global Katz scores.

(d) Terminating residual versus precision for personalized Katz scores.

Figure 3.5: Performance versus required precision for Katz Centrality on directed graphs (with $\alpha = 0.9/\|A\|_2$).

relatively constant speedup for precisions of 0.5-0.9 and then a sharp drop in speedup for a precision of 1.0. This suggests that while there are vertices in the ranking vector with these necessary gaps to guarantee ranking, in order to find the gap between successive vertices the solver needs to reach a high level of accuracy. From this we can conclude that if the use case can tolerate a few false positives in the set of the top R highly ranked vertices, then we can obtain the top vertices in a graph quickly with relatively high precision.

Next we analyze the effect of our stopping criterion on PageRank. Here we use the theory from Theorem 2 for both undirected (Figure 3.6) and directed (Figure 3.7)



(a) Speedup versus precision for global PageRank scores.

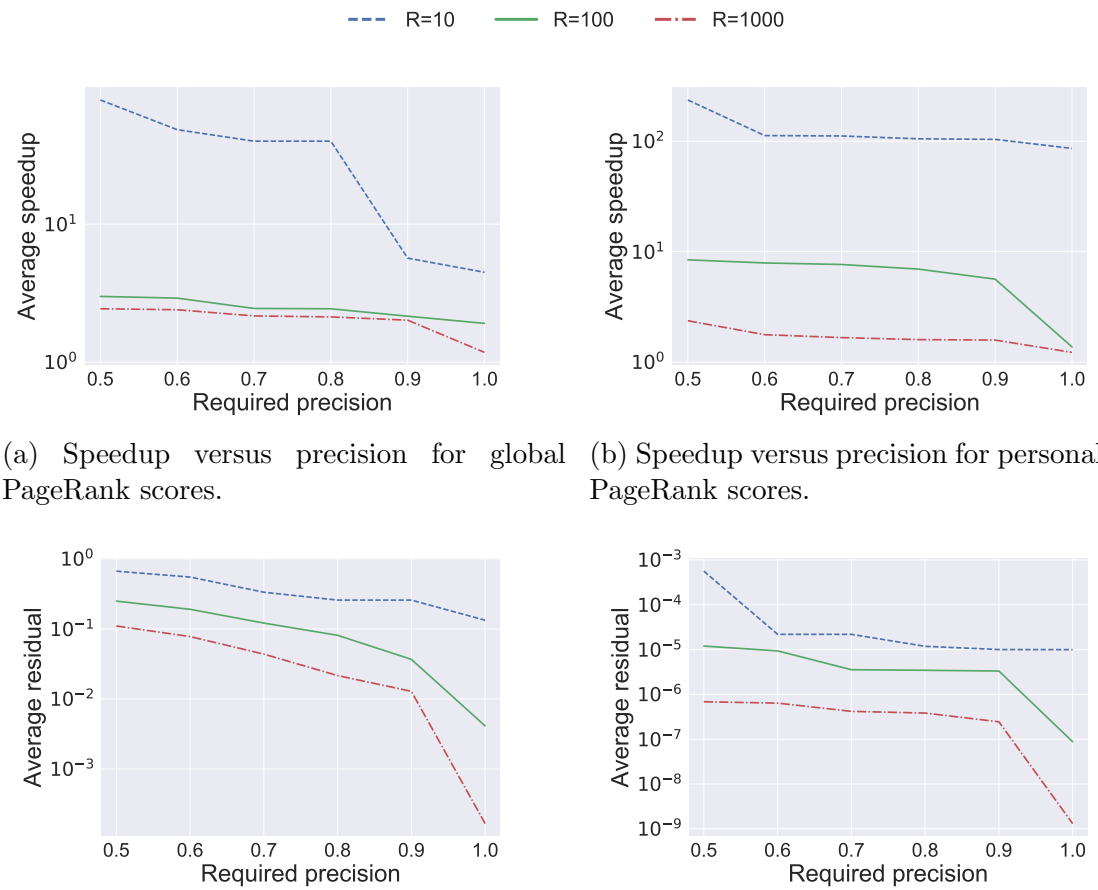
(b) Speedup versus precision for personalized PageRank scores.

(c) Terminating residual versus precision for global PageRank scores.

(d) Terminating residual versus precision for personalized PageRank scores.

Figure 3.6: Performance versus required precision for PageRank on undirected graphs.

graphs. Similar to the results for Katz Centrality earlier, we see higher speedups and lower terminating residuals for the personalized results (Figures 3.6b and 3.6d) compared to their global counterparts (Figures 3.6a and 3.6c). For PageRank, however, the speedups in the personalized case are considerably higher than the respective global ones. We also see similar trends of larger speedups and higher terminating residuals for smaller values of R. Note that in Figures 3.6a and 3.6b there are regions in the plot where the speedup for R=10 is less than the speedup for R=100 (for the same precision). This is likely due to the behavior of the ranking vector for these values. For example, if the centrality values of vertices in the top 10-20 vertices are very similar, our stopping criterion would have to iterate further in order to obtain



(a) Speedup versus precision for global PageRank scores.

(b) Speedup versus precision for personalized PageRank scores.

(c) Terminating residual versus precision for global PageRank scores.

(d) Terminating residual versus precision for personalized PageRank scores.

Figure 3.7: Performance versus required precision for PageRank on directed graphs.

that required gap of $2\epsilon_k$. Likewise, if the values for vertices 100 and 101 are very far apart and the gap is found almost immediately, then the stopping criterion will be able to terminate sooner. This behavior of the centrality vector would lead to cases where speedup for higher values of R is greater than that of lower values of R. Finally, we examine our stopping criterion on PageRank for directed graphs. Like Katz Centrality on directed graphs, the terminating residual (both global and personalized rankings) stays relatively constant for a required precision between 0.5-0.8 or 0.9 and then sharply drops for a required precision of 1.0.

3.2.3 Perfect ordering of top

We have shown that we are successfully able to efficiently identify sets of top ranked vertices in networks for various set sizes. Experimentation shows that the theory is sound across several real-world networks. While the previous experiments are only concerned with returning the top set of vertices, here we impose the additional constraint of perfect ordering of this set. We not only want the most highly ranked vertices, but we also want them in the correct ordering as given by the exact solution. We motivate the next experiment with an example concerning the web-Google graph described earlier. When entering a search term into the Google search engine, a typical user will only traverse the first few pages of search results, about 75-100 total pages, expecting most relevant results to be at the top of the list. In this use case, it is important to ensure the ordering of these results is correct. We are able to apply the theory from Theorem 1 in this application and provide a guarantee on how many vertices we can accurately certify are in the correct ordering in the top of the ranking vector compared to the exact solution using Katz Centrality. In this case, we look at the gaps between successive vertices i and $i+1$ to ensure each pairwise comparison of vertices has the necessary gap to prove the correctness of the relative ordering. Running a solver to machine precision to identify top sets in networks cannot in fact provide any theoretical guarantee of how many vertices in the approximation are in the correct ordering compared to the exact solution. In this experiment, we are interested in finding P such that $P = argmax_i |d_i^{(k)} - d_{i+1}^{(k)}| > 2\epsilon_k$, where P is the number of vertices in the top of the vector in the correct order compared to the exact solution. We traverse the sorted ranking vector $\mathbf{d}^{(k)}$ after 10 iterations of conjugate gradient to find the first place where the gap of $2\epsilon_k$ is not satisfied. When this occurs, we know that the previous vertices are in the correct ordering since each pair-wise comparison of previous vertices satisfied the gap.

Figure 3.8 plots the distribution of P values for both undirected and directed

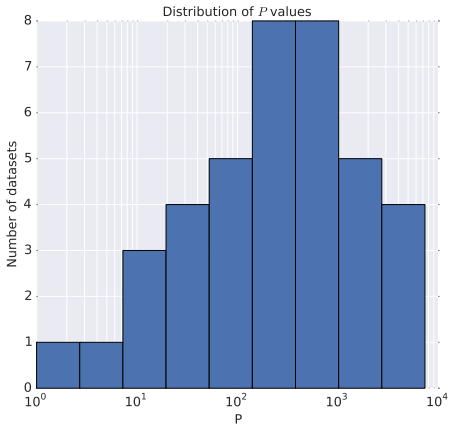
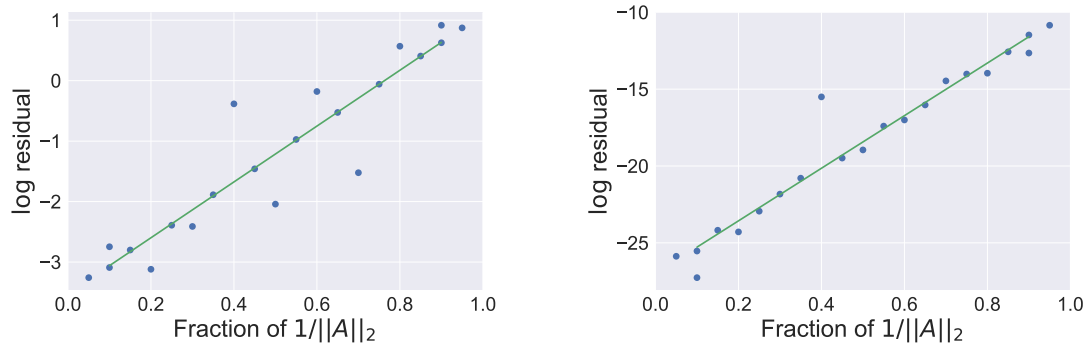


Figure 3.8: Histogram of P values for different networks.

networks using both Katz Centrality and PageRank, with values of 0 omitted. Note the x-axis is on a log-scale. In most cases, we are able to accurately certify at least hundreds of vertices, with an average across all datasets of $P = 903$. For cases where we are only able to guarantee 1 or 0 vertices, we offer a possible explanation. If there are vertices with the same ranking at the top of the exact solution, our theory will not be able to go beyond this point because the necessary gap does not exist. Regardless, from a data analysis standpoint, the numbers of vertices able to be accurately certified in the exact order in the top validate our theory being used in this use case. Our ability to accurately certify hundreds of vertices in the correct order is very applicable.

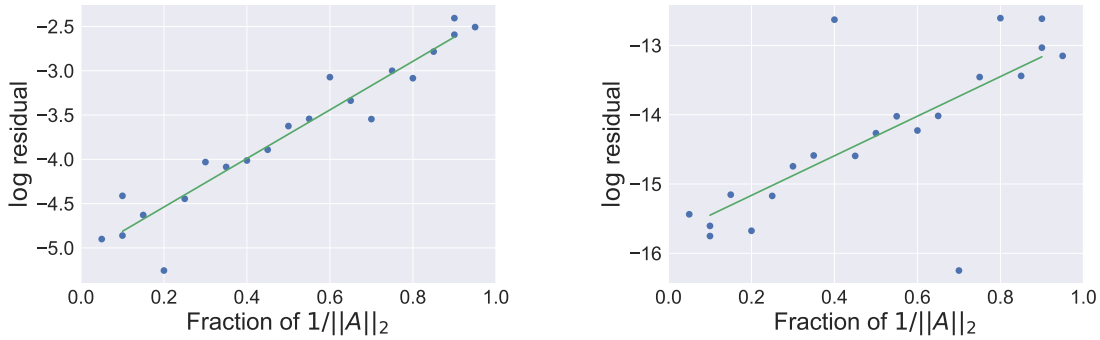
3.2.4 Effect of stopping criterion on harder problems

Finally we investigate on what problems our method proves to be the most useful. For these results, we focus our analysis exclusively on Katz Centrality. We know as $\alpha \to \frac{1}{\|A\|_2}$, the problem becomes more ill-conditioned and typically requires more iterations to converge to machine precision. Since $\alpha \in (0, 1/\|A\|_2)$, we apply our stopping criterion to the different graphs for various α in this range. Figure 3.9 plots the relationship between α and the residual norm obtained when the solver terminates using our criterion for undirected graphs for global (Figure 3.9a) and personalized



(a) Global Katz scores. (b) Personalized Katz scores.

Figure 3.9: Terminating residual obtained as we increase α for Katz scores in undirected graphs.



(a) Global Katz scores. (b) Personalized Katz scores.

Figure 3.10: Terminating residual obtained as we increase α for Katz scores in directed graphs.

(Figure 3.9b) rankings. The blue scatterplot points show the averaged values and the green line in the plots is a line fitted using regression analysis. We use values of $\alpha \in \{\frac{.05}{\|A\|_2}, \frac{.1}{\|A\|_2}, \cdots, \frac{.95}{\|A\|_2}\}$. For each value of α, the log of the averaged residual norm obtained upon termination using our stopping criterion is plotted across graphs. All results are averaged over values of $R = 10$, 100, and 1000 and over all the graphs. When running to machine precision, the residual norm upon termination is typically $r_k \approx 10^{-15}$, but we see that we never have to iterate until machine precision using our new stopping criterion if we are interested in only the top vertices in a graph. Regression analysis of these results shows a strong linear correlation with a slope of

4.617 and mean sum of squares of 0.724 for the global values and a slope of 17.110 and mean sum of squares of 0.862 for the personalized values. We repeat the same analysis for the directed networks in Figure 3.10, with the global results plotted in Figure 3.10a and the personalized results plotted in Figure 3.10b. The slope of the line plotted for the global results is 2.74 with a mean sum of squares of 0.804 and the slope for the personalized results is 2.86 with a mean sum of squares of 0.544. The linear relationship suggests that we need less accurate approximate solutions for harder problems as $\alpha \to \frac{1}{\|A\|_2}$ to obtain the top vertices in the graph. Typically the harder problems tend to take thousands of iterations to converge with the standard stopping criterion of iterating until a residual norm of 10^{-15}, but with our stopping criterion we can converge faster at a lower tolerance to solve the desired data mining task for the global scores. The low residual norm suggests we are able to certify the top R correctly with low fidelity solutions and we are able to use this technique to turn harder linear algebra problems into easier data mining problems.

3.3 Conclusions

This work bridged the two research areas of numerical accuracy of solvers and network analysis by understanding how the error in a solver affects the data analysis problem of ranking. By treating the problem of ranking vertices in a graph as understanding numerical accuracy in a linear solver, we presented how the error in the numerical problem affects the solution to the original data analysis problem of ranking. Our aim in this work was to provide theoretical guarantees to bound the error in an approximate solution from an iterative method to the exact Katz Centrality and PageRank scores of vertices in a network. We certified ranking in undirected and directed graphs using global and personalized Katz and PageRank scores. We turned the data analysis problem of ranking vertices in graph into the numerical problem of understanding accuracy in a linear solver. This allowed us to provide guarantees as to

how accurate of a solution to the numerical problem we need to certify highly ranked vertices in graphs. Using our theoretical guarantees we were able to identify the most central vertices with either Katz Centrality or PageRank with high confidence. We do not need to accurately compute the centrality scores for every vertex and therefore could reduce computation time. Using the theory and error analysis, we developed a new stopping criterion that can be used in conjunction with any iterative solver to determine when to terminate given a desired number of highly ranked vertices with some preset precision, where the precision provides a bound on how many false positives we will tolerate being returned. The result of our analysis is a reduction in the number of iterations taken to solve the data analysis problem of ranking in graphs while maintaining a high precision rate in identifying top vertices. In fact, for personalized PageRank scores we obtained speedups of several orders of magnitude. We demonstrated this on several real-world networks, giving high confidence that the important portion of the ranking is correct. We presented experiments validating the theory as a stopping criterion that can be used in conjunction with any iterative solver, leading to significant algorithmic improvements. When using the theory to identify top ranked vertices we were able to do so with very few false positives. Finally, we also showed perfect recall of the top vertices with respect to the exact solution is possible with our theory. As evidenced by the close relationship between the theory for Katz Centrality and PageRank, the results from this section can be applied to any linear solver based ranking. Identifying highly ranked vertices by Katz Centrality or PageRank are just two examples in practice presented in this work, but the theory is generalizable to other linear algebra based ranking metrics.

CHAPTER 4
DYNAMIC ALGORITHMS FOR CENTRALITY MEASURES

This chapter presents several algorithms for updating different centrality metrics in dynamic graphs. Given an analytic and a dynamic graph, a naive method of obtaining an updated metric is to recompute the metric from scratch every time the graph changes. However, this becomes extremely computationally infeasible as the graph grows larger and more and more changes are applied to the graph. We therefore seek to update analytics efficiently for dynamic graphs without needing to perform a full static recomputation. Sections 4.1 and 4.2 present dynamic algorithms for updating Katz Centrality from 1) a linear algebraic perspective [6] and 2) an agglomerative graph-based method [8], respectively. Section 4.3 presents an algorithm for updating nonbacktracking walk-based centrality scores on dynamic graphs and Section 4.4 presents a method for efficiently updating matrix exponential-based centrality scores for dynamic graphs [9]. For all methods presented, we show our dynamic algorithm is faster than naive static recomputation and demonstrate that the quality of our method is on par with that of the corresponding static method. In many cases we see several orders of magnitude of speedup comparing our method to static recomputation, indicating that our algorithms are faster and more efficient when applied to dynamic graphs.

4.1 Dynamic Katz Centrality using Linear Algebra

In this section, we present a new method from a linear algebraic standpoint to incrementally update Katz Centrality scores in a dynamic graph. Our algorithm is faster than recomputing centrality scores from scratch every time the graph is updated and returns high quality results that are similar to results obtained with a simple static

recomputation method. We additionally present an alternate approach and discuss its shortcomings compared to our algorithm. We examine how our algorithm behaves with respect to both global and personalized centrality scores and analyze how the granularity of the time step affects the quality of our algorithm. We compare our dynamic algorithm to multiple static recomputation methods and also examine the effect of our algorithm if we are only concerned with recall of the highly ranked vertices in dynamic graphs. Section 4.1.1 provides the necessary background and definitions required to understand our work. In Section 4.1.2 we present the alternate method and provide the motivation for our dynamic algorithm. We present our algorithm for updating Katz Centrality in dynamic graphs in Section 4.1.3. Section 4.1.4 provides an analysis of our method on both synthetic and real-world networks with respect to performance and quality. In Section 4.1.5 we discuss a possible approach for handling vertex additions and deletions and in Section 4.1.6 we conclude.

4.1.1 Background & Definitions

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

Recall we denote global Katz scores as $A(I-\alpha A)^{-1}\mathbf{1}$ and personalized Katz scores w.r.t. a seed vertex i as $A(I-\alpha A)^{-1}\mathbf{e}_i$. For both cases, the result is an n-length vector. In the global case, the ith value in the vector represents the total number of weighted walks of all lengths starting at vertex i and in the personalized case the ith

value in this vector represents the number of weighted walks of all lengths ending at vertex i. We set $\alpha = 0.85/\|A\|_2$ as in [21], and in this work we study both global and personalized scores.

As mentioned before, since directly solving for the exact Katz Centrality scores $\mathbf{c}$ is computationally infeasible and quickly becomes very expensive and impractical as n grows large, in practice we use iterative methods to obtain an approximation which costs $\mathcal{O}(m)$ provided the number of iterations is not very large. Unless otherwise stated, all the work here assumes a starting approximation $\mathbf{x}^{(0)}$ as the all zeros vector, although any starting vector can be chosen to initialize the iterative solver. The residual at the kth iteration is defined as $\mathbf{r}^{(k)} = \mathbf{b} - M\mathbf{x}^{(k)}$. We let $M = I - \alpha A$, so we solve the linear system $M\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$ using an iterative method and then obtain the Katz scores using a matrix-vector multiplication in $\mathcal{O}(m)$ as $\mathbf{c} = A\mathbf{x}$. We set $\mathbf{b} = \mathbf{1}$ for the global scores and $\mathbf{b} = \mathbf{e}_i$ for the personalized scores. The iterative method we use here is the Jacobi algorithm [77] outlined in Algorithm 1. Here, D is the matrix consisting of the diagonal entries from M and R is the matrix of all off-diagonal entries of M. We terminate the solver when the solution changes by less than a fixed tolerance tol [17], or when $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_2 < tol$.

Our dynamic algorithm is also motivated by principles of iterative refinement, another iterative method that adds a correction to the current guess to obtain a more accurate approximation [78]. To compute the solution $\mathbf{x}$ to the linear system $M\mathbf{x} = \mathbf{b}$, iterative refinement repeatedly performs the following steps at each iteration k.

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

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

4.1.2 Motivation & Initial Approach

Static Algorithm

Given edge updates to the graph, the static algorithm to recompute the Katz Centrality scores in the updated graph first calculates $\mathbf{x}$ from scratch using an iterative method and then calculates $\mathbf{c}$ using a single matrix-vector multiplication. This procedure is given in Algorithm 6 to obtain the new solution $\mathbf{c}_{t+1}$ at time $t+1$ given updates ΔA to the graph. After a batch of edges has been inserted into the network, the adjacency matrix is updated to A_{t+1} and the vector $\mathbf{x}_{t+1}$ is recomputed using the Jacobi method from Algorithm 1.

Algorithm 6 Solve for $\mathbf{c}_{t+1}$ at time $t+1$ given new edge updates ΔA.

```
1: procedure STATIC_KATZ(A_t, ΔA)
2:     A_{t+1} = A_t + ΔA                          ▷ Updated adjacency matrix
3:     M_{t+1} = I − αA_{t+1}                      ▷ New linear system
4:     x_{t+1} = JACOBI(M_{t+1}, 1, 10^{−4})       ▷ Recomputed vector
5:     c_{t+1} = A_{t+1} x_{t+1}                   ▷ New Katz scores
6: return c_{t+1}
```

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

Motivation

In many low-latency applications, the number of edge updates, or equivalently, the size of ΔA, is significantly smaller than the size of the entire graph A. If the change

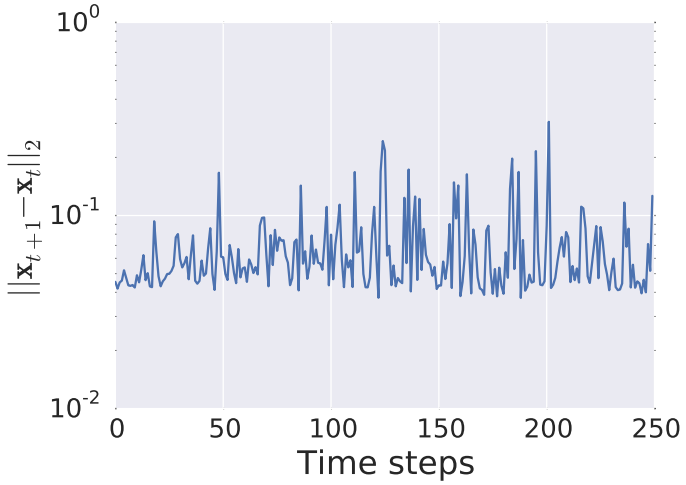


Figure 4.1: Difference in consecutive solutions over time. Small changes in solutions suggest a dynamic algorithm could work by applying incremental updates to previous solutions.

ΔA is small relative to the size of the graph, the new graph will be similar to the old graph. It follows that the new solution $\mathbf{x}_{t+1}$ at time $t+1$ might be similar to the old solution $\mathbf{x}_t$ at time t. This is the intuition behind our dynamic algorithm. Figure 4.1 plots the differences between subsequent solutions for global scores each time the graph changes for the FACEBOOK graph (63,731 vertices and 817,035 edges). The x-axis simulates time as more edges are being added into the graph. We insert 1000 edges into the graph at each time step. The y-axis is the 2-norm difference between solutions at consecutive timepoints, $\|\mathbf{x}_{t+1} - \mathbf{x}_t\|_2$. Since the Katz scores themselves can be as high as 10^4, a difference of 10^{-1} across insertions of edges over time is relatively small. This indicates that the solutions themselves are not very different, suggesting that the static algorithm of recomputing the centrality metric from scratch is doing a lot of unnecessary work. Our dynamic algorithm therefore only targets places in the vector that are affected by updates to the graph and obtains the new solution $\mathbf{x}_{t+1}$ by solving for a correction $\mathbf{\Delta x}$ to add to the old solution $\mathbf{x}_t$ to calculate $\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{\Delta x}$.

Initial Approach

Here we present a "first-pass" algorithm and discuss its shortcomings. This provides the motivation for the development of our dynamic algorithm in Section 4.1.3. Suppose we have the solution $\mathbf{x}_t$ for the adjacency matrix A_t at a specific timepoint t. We want to solve for the new solution at time $t+1$ as $\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{\Delta x}$. Given edge updates to the graph, we want to solve for the vector $\mathbf{x}_{t+1}$ in the linear system $(I - \alpha A_{t+1})\mathbf{x}_{t+1} = \mathbf{1}$ for the global scores, or $(I - \alpha A_{t+1})\mathbf{x}_{t+1} = \mathbf{e}_i$ for the personalized scores equivalently. Using basic algebra we can rearrange the terms in the linear system to derive an iterative update as follows:

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

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

$$\mathbf{\Delta x} = \alpha A_{t+1}\mathbf{\Delta x} + \alpha \Delta A\mathbf{x}_t, \tag{4.1}$$

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

$$\mathbf{\Delta x}^{(k+1)} = \alpha A_{t+1}\mathbf{\Delta x}^{(k)} + \alpha \Delta A\mathbf{x}_t \tag{4.2}$$

However, this simplistic approach tends to accumulate error over time instead

of converging to the same solution as static recomputation. We provide a more in-depth analysis of the quality of this alternate method in Section 4.1.4. This approach (henceforth referred to as the "alternate" method) is based off of a forward error analysis. Therefore, we next present our dynamic algorithm based off of a backward error analysis in Section 4.1.3.

4.1.3 Dynamic Algorithm

Our dynamic algorithm computes the correction $\mathbf{\Delta x}$, the difference in the solutions at timepoints t and $t+1$, using principles of iterative refinement. For the purposes of deriving the algorithm, we do so w.r.t. the global scores. For personalized scores w.r.t. vertex i, we simply replace the vector $\mathbf{1}$ with $\mathbf{e}_i$. Since we use the old solution as a starting point for the new solution, we first measure how close the old solution is to solving the system for the new graph. We do so by introducing the concept of an "approximate residual" denoted as $\widetilde{\mathbf{r}}_{t+1}$. This can be written in terms of the current residual at time t, $\mathbf{r}_t = \mathbf{1} - M_t\mathbf{x}_t$, edge updates ΔA, and the old solution $\mathbf{x}_t$. The algorithm to compute $\widetilde{\mathbf{r}}_{t+1}$ is given in Algorithm 7 with the corresponding proof of correctness in Theorem 4.

Algorithm 7 Solve for approximate residual $\widetilde{\mathbf{r}}_{t+1}$ at time $t+1$.

```
1: procedure GET_APPROXIMATE_RESIDUAL(ΔA, r_t, x_t)
2:     r̃_{t+1} = r_t + αΔA x_t
3: return r̃_{t+1}
```

Theorem 4. *Algorithm 7 correctly calculates the approximate residual at time $t+1$.*

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

to solving the updated system A_{t+1}.

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

□

We then use the approximate residual $\widetilde{\mathbf{r}}_{t+1}$ to solve a linear system for the correction $\mathbf{\Delta x}$. Solved exactly, this linear system will give the same scores as static recomputation but solved to some preset tolerance as discussed earlier, it will provide a good quality approximation of the updated centrality scores. We examine the effect of varying the tolerance on the performance of our dynamic algorithm in Section 4.1.4. This procedure and the corresponding proof of correctness are given in Algorithm 8 and Theorem 5 respectively.

Algorithm 8 Use iterative refinement to obtain $\mathbf{\Delta x}$.

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

Theorem 5. *Algorithm 8 correctly calculates the correction* $\mathbf{\Delta x}$ *at time* $t + 1$.

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

using principles of iterative refinement.

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

$$\mathbf{\Delta x} - \alpha A_{t+1}\mathbf{\Delta x} = \mathbf{r}_t + \alpha \Delta A \mathbf{x}_t$$

We can turn this into an iterative update:

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

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

The final step of our algorithm is to update the residual $\mathbf{r}_t$ for the next timepoint. We do so by calculating $\mathbf{\Delta r}$, the difference in the two residuals at time t and $t+1$. This procedure is given in Algorithm 9 with the corresponding proof of correctness in Theorem 6.

Algorithm 9 Updating residual at time $t+1$.

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

Theorem 6. *Algorithm 9 correctly updates the residual at time $t+1$.*

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

difference between the two as $\mathbf{\Delta r}$.

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

□

The entire procedure for updating Katz Centrality scores in a dynamic graph is outlined in Algorithm 10, DYNAMIC_KATZ, and uses the three previously described subroutines. First in line 2 we calculate the current residual $\mathbf{r}_t$, which is easily obtained given the current snapshot of the graph A_t and solution $\mathbf{x}_t$ at time t. In line 3, we form the new snapshot of the graph A_{t+1} using the new batches of edges that are being inserted into the graph, In line 4 we call the first subroutine GET_APPROXIMATE_RESIDUAL, Algorithm 7, to return the approximate residual $\widetilde{\mathbf{r}}_{t+1}$. Next in line 5 we solve for the difference $\mathbf{\Delta x}$ between the vectors $\mathbf{x}_{t+1}$ and $\mathbf{x}_t$ using the subroutine OBTAIN_DEL_X, Algorithm 8. In line 6 we calculate the new solution $\mathbf{x}_{t+1}$ using the old solution $\mathbf{x}_t$ and the calculated correction $\mathbf{\Delta x}$. Finally, after updating the solution from time t to the solution at $t+1$, lines 6 and 8 update the residual between these two timepoints using the subroutine UPDATE_RESIDUAL in Algorithm 9. Finally, at the end of the procedure in line 9 we return the new solution $\mathbf{x}_{t+1}$.

Note that while in this section we only examine edge insertions in a dynamic

Algorithm 10 Solve for $\mathbf{x}_{t+1}$ at time $t+1$ given previous solution $\mathbf{x}_t$ at time t and new edge updates ΔA.

procedure DYNAMIC_KATZ($A_t, \mathbf{x}_t, \Delta A$)
  $\mathbf{r}_t = \mathbf{1} - (I - \alpha A_t)\mathbf{x}_t = \mathbf{1} - \mathbf{x}_t + \alpha A_t$
  $A_{t+1} = A_t + \Delta A$
  $\widetilde{\mathbf{r}}_{t+1}$ = GET_APPROXIMATE_RESIDUAL($\Delta A, \mathbf{r}_t, \mathbf{x}_t$)
  $\mathbf{\Delta x}$ = OBTAIN_DEL_X($A_{t+1}, \widetilde{\mathbf{r}}_{t+1}$)
  $\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{\Delta x}$
  $\mathbf{\Delta r}$ = UPDATE_RESIDUAL($A_{t+1}, \Delta A, \mathbf{x}_{t+1}$)
  $\mathbf{r}_{t+1} = \mathbf{r}_t + \mathbf{\Delta r}$
return $\mathbf{x}_{t+1}$

graph, the algorithm is equally well suited for handling edge deletions. Here, all nonzero values in ΔA corresponding to edge insertions are set to 1 but edge deletions can be handled easily by setting the corresponding value in ΔA to -1 as described in Section 2.1.

Complexity Analysis

The majority of the work done by the dynamic algorithm is in Algorithm 8 (OBTAIN_DEL_X). Since we still require a matrix-vector multiplication by A_{t+1} at the end of the algorithm, the worst-case complexity of the dynamic algorithm is the same as static recomputation and is $\mathcal{O}(m)$, apart from a constant (based on the number of iterations taken by the iterative solver). However, in practice we observe that we are able to obtain significant speedups in both time and iterations compared to static recomputation while maintaining a good quality of results returned. This is due to the fact that the number of iterations taken by our dynamic algorithm is far fewer than that of static recomputation and we are able to converge to the solution faster.

4.1.4 Results

We test our method of updating Katz Centrality scores in dynamic graphs on both synthetic and real-world networks. For synthetic networks, we use Erdos-Renyi [79]

and R-MAT graphs [80]. In the Erdos-Renyi model, a graph is constructed by connecting vertices randomly. All edges have the same probability for existing in the graph. An R-MAT generator creates scale-free networks designed to simulate real-world networks. Consider an adjacency matrix: the matrix is subdivided into four quadrants, where each quadrant has a different probability of being selected. Once a quadrant is selected, this quadrant is recursively subdivided into four subquadrants and using the same probabilities, we select one of the subquadrants. This process is repeated until we arrive at a single cell in the adjacency matrix. An edge is assigned between the two vertices making up that cell. For real-world networks, we draw from the KONECT collection of datasets [76]. The five datasets used are given in Table 4.1 and comprise a mixture of citation and social networks. These graphs are chosen because they have timestamps associated with the edges to represent temporal data. The code was implemented in Python.

Table 4.1: Graphs used in experiments. Columns are graph name, number of vertices, and number of edges.

Graph	**|V|**	**|E|**
facebook	63,731	817,035
gowalla	196,591	950,327
dblp	317,080	1,049,866
dogster	426,820	8,546,581
youtube	1,134,890	2,987,624

To have a baseline for comparison, we treat scores obtained from static recomputation as ground truth. Every time we update the centrality scores using our dynamic algorithm, we recompute the centrality vector statically using Algorithm 6. Denote the vector obtained by static recomputation by $\mathbf{x}_S$ and the vector obtained by our dynamic algorithm by $\mathbf{x}_D$. We create an initial graph G_0 using the first half of edges, which provides a starting point for both the dynamic and static algorithms. To simulate a stream of edges in a dynamic graph, we insert the remaining edges in batches of size b and apply both algorithms. For the synthetic graphs, the edges are permuted

randomly during insertion. Edges in real graphs are inserted in timestamped order. We use batch sizes of $b = 1, 10, 100$, and 1000 and vary the tolerance to which we solve for in Algorithm 1 (the Jacobi method) and provide analysis on how this affects the results of our algorithm.

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

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

$$speedup_{time} = \frac{T_S}{T_D}.$$

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

Calculate the speedup w.r.t. the number of iterations as

$$speedup_{iter} = \frac{I_S}{I_D}.$$

Table 4.2: Speedup in time for Erdos-Renyi graphs.

Average degree	**10**	**20**	**30**	**40**	**50**
$n = 1024$	1.44×	1.62×	1.8×	1.99×	2.17×
$n = 2048$	1.51×	1.77×	2.0×	2.25×	2.49×
$n = 4096$	1.66×	2.03×	2.37×	2.85×	3.34×
$n = 8192$	1.95×	2.55×	3.05×	4.02×	5.09×
$n = 16384$	2.51×	3.5×	4.34×	6.0×	8.02×

Table 4.3: Speedup in iterations for Erdos-Renyi graphs.

Average degree	**10**	**20**	**30**	**40**	**50**
$n = 1024$	4.56×	5.01×	5.37×	5.71×	5.99×
$n = 2048$	4.82×	5.4×	5.82×	6.17×	6.5×
$n = 4096$	5.05×	5.77×	6.27×	6.7×	7.1×
$n = 8192$	5.25×	6.12×	6.69×	7.24×	7.73×
$n = 16384$	5.40×	6.42×	7.04×	7.73×	8.33×

Table 4.4: Speedup in time for R-MAT graphs.

Average degree	**10**	**20**	**30**	**40**	**50**
$n = 1024$	1.75×	1.95×	2.15×	2.44×	2.7×
$n = 2048$	1.98×	2.39×	2.7×	3.14×	3.56×
$n = 4096$	2.42×	3.12×	3.62×	4.3×	5.08×
$n = 8192$	3.35×	4.32×	5.25×	6.41×	7.46×
$n = 16384$	4.63×	6.26×	7.64×	9.15×	10.46×

Tables 4.2 and 4.3 give the average speedup in time and reduction in iterations respectively for Erdos-Renyi graphs, and Tables 4.4 and 4.5 show the same values for R-MAT graphs. As we increase the average degree for both types of graphs, the speedups in time and iterations are larger. Additionally, we see greater speedups for graphs with larger values of n. The dynamic algorithm likely has more of an effect for larger graphs because there is more work to be done for larger graphs with the static

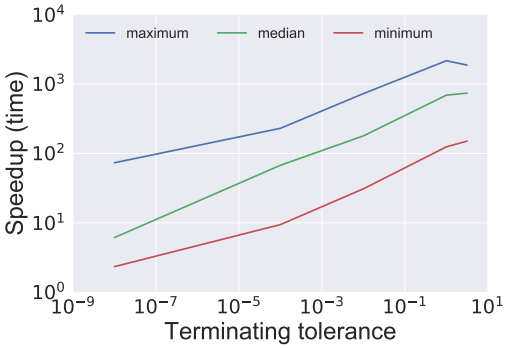
Table 4.5: Speedup in iterations for R-MAT graphs.

Average degree	**10**	**20**	**30**	**40**	**50**
$n = 1024$	4.89×	5.29×	5.6×	6.01×	6.38×
$n = 2048$	5.12×	5.77×	6.27×	6.73×	7.12×
$n = 4096$	5.34×	6.2×	6.69×	7.24×	7.66×
$n = 8192$	5.81×	6.52×	7.18×	7.77×	8.25×
$n = 16384$	6.0×	6.89×	7.62×	8.29×	8.72×

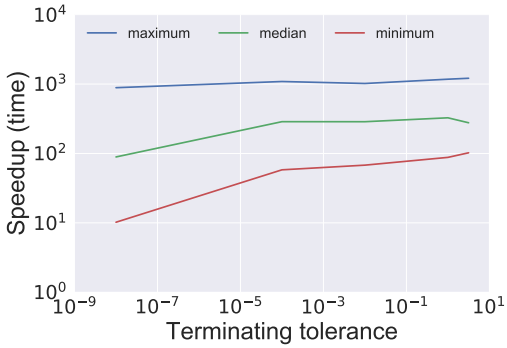
algorithm. Unlike the static algorithm, our dynamic algorithm only traverses parts of the graph where updates have occurred. These trends persist for both Erdos-Renyi and R-MAT graphs, but typically we find that R-MAT graphs have greater speedups than their respective Erdos-Renyi counterparts.

Next we examine the performance of our algorithm on the real-world graphs. First we look at the effect of the terminating tolerance on the speedup (in both time and iterations) obtained in Figure 4.2. Specifically, Figures 4.2a and 4.2b plot the speedup in time for global and personalized scores respectively and Figures 4.2c and 4.2d plot the speedup in iterations for global and personalized scores respectively. Results are averaged across the five real datasets and show maximum (in blue), median (in green), and minimum (in red) speedups. Note that the y-axis in Figures 4.2a and 4.2b is on a log scale with base 10 and the y-axis in Figures 4.2c and 4.2d is on a log scale with base 2 for clarity.

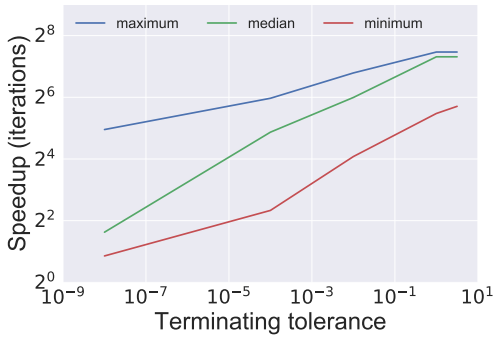
For the global scores, we observe that as the increase the value of the tolerance to which we solve for, we obtain greater speedups. This intuitively makes sense because as we increase the value of the tolerance required to terminate (meaning a less accurate solution will suffice), the iterative solver will take fewer iterations to converge and our dynamic algorithm will have more of an effect. For the personalized scores, we see more of a plateau and the speedups obtained seem to be independent of the preset tolerance. This is likely due to the fact that the personalized scores themselves are so small. Therefore, it may take the same number of iterations to converge to a tolerance of at least 10^{-1} as it does to converge to 10^{-3} for example, so we see very



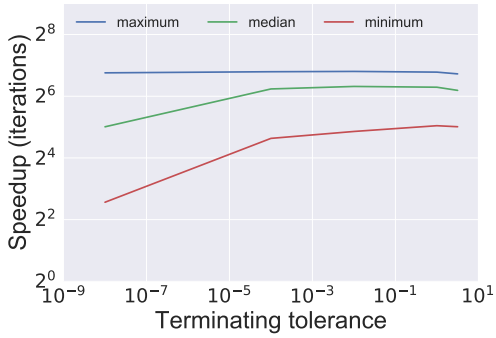
(a) Speedup (time) for global scores.



(b) Speedup (time) for personalized scores



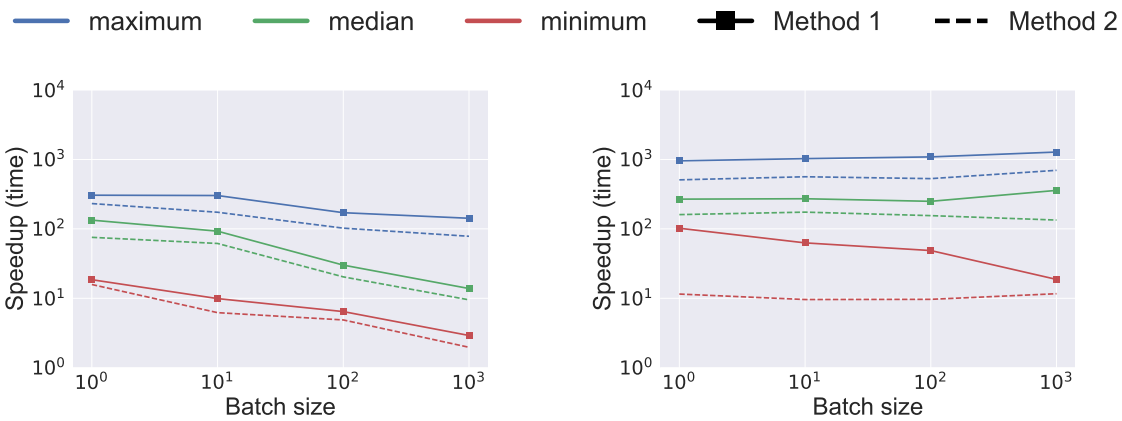
(c) Speedup (iterations) for global scores.



(d) Speedup (iterations) for personalized scores.

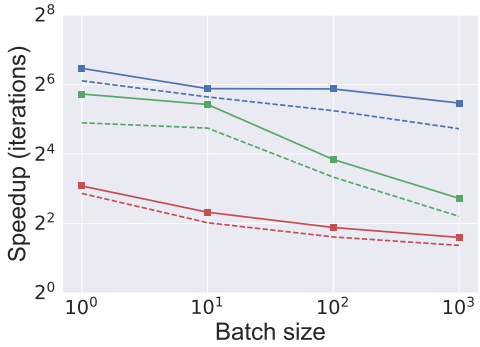
Figure 4.2: Speedup (time and iterations) versus tolerance. Higher is better.

little differences in the speedups for these tolerances. We also note that the speedups (in both time and iterations) for the personalized scores are greater than their global counterparts. Since the values of the scores are so small in the personalized case, the iterative solver takes more total iterations to converge and the dynamic algorithm has more of an effect here. Nevertheless, overall we obtain speedups of several orders of magnitude and for the global scores on average about 100× speedup in time and 32× speedup in iterations. Similarly for the personalized scores, we obtain on average about a 200× speedup in time and about a 64× speedup in iterations. Even for very low values of the tolerance (such as 10^{-8}), we always obtain $> 1\times$ speedup. This indicates we can obtain fairly accurate scores, and with our method do so much faster than static recomputation.

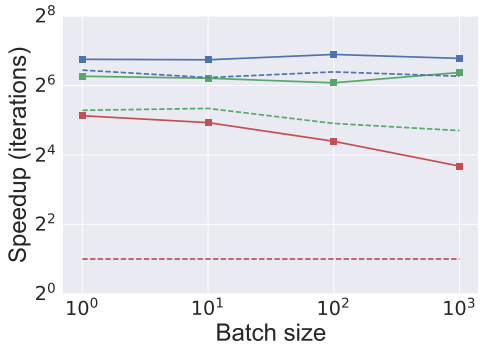


(a) Speedup in time for global scores.

(b) Speedup in time for personalized scores.



(c) Speedup in iterations for global scores.



(d) Speedup in iterations for personalized scores.

Figure 4.3: Speedup (time and iterations) versus batch size. Higher is better.

Next we examine the speedups obtained as a function of batch size and compare our dynamic algorithm against two different static methods in Figure 4.3. Both static methods evaluate $\mathbf{x}_S$ using Algorithm 6 but start with different initial starting vectors in line 3 in Algorithm 1 (JACOBI).

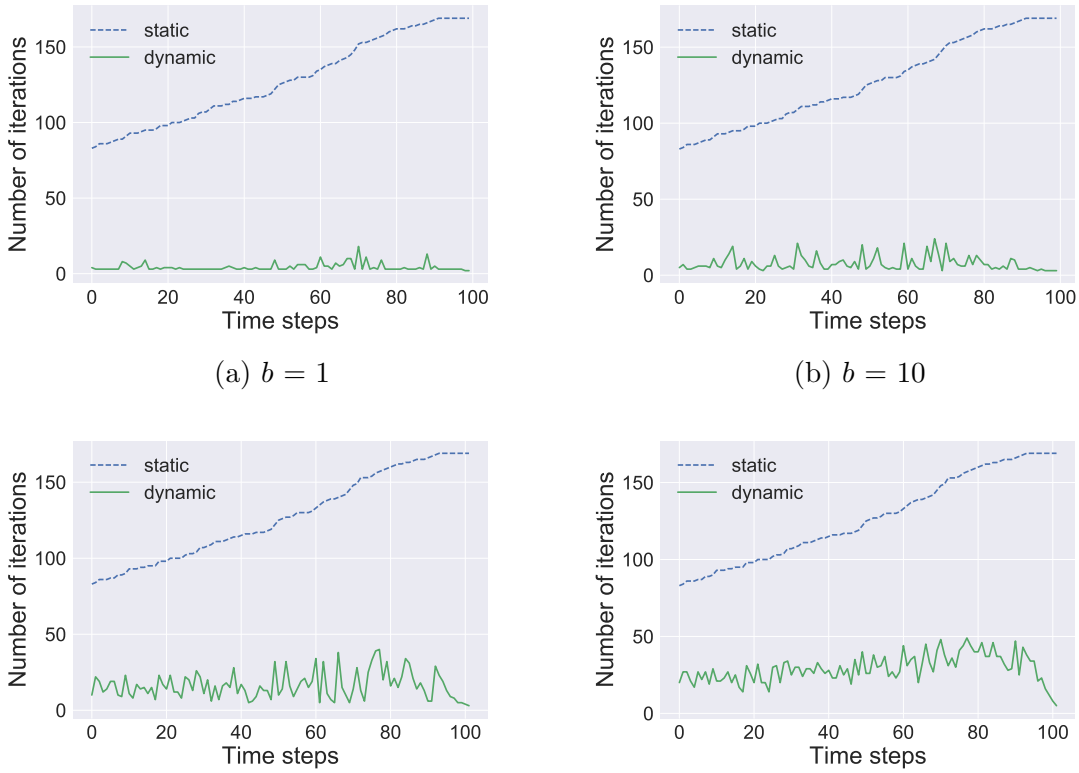
1. Method 1: uses an initial starting vector of $\mathbf{x}^{(0)} = \mathbf{0}$.

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

Figures 4.3a and 4.3b plot the speedup in time versus batch size for global and personalized scores respectively, comparing our dynamic algorithm against static re-

computation. Similarly, Figures 4.3c and 4.3d plot the speedup in iterations versus batch size for global and personalized scores respectively. We show the maximum, median, and minimum speedup averaged over the 5 real graphs. Method 1 is plotted with a solid line with squares and Method 2 is plotted with a dotted line. For this, we examine results only for a terminating tolerance of 10^{-4} although the trends observed for other tolerances are similar. Note again that the y-axis in Figures 4.3a and 4.3b is on a log scale with base 10 and the y-axis in Figures 4.3c and 4.3d is on a log scale with base 2. In Figure 4.3a we see that our dynamic algorithm can be over two orders of magnitude faster for a batch size of 1 than both static recomputation approaches. It is expected that Method 2 is faster than Method 1, since we initialize Jacobi with the vector $\mathbf{x}_t$ that is likely closer to the new solution $\mathbf{x}_{t+1}$ than $\mathbf{0}$, but our dynamic algorithm is still able to outperform this method in both time and iterations. The median speedup in time for the global scores is about 100× for a batch size of 1 and about 200× for the personalized scores for a batch size of 1. Even for a batch size of 1000 edges we always have greater than a 1× speedup. Figure 4.3c shows that we can obtain over an 80× reduction in iterations for both global and personalized scores for a batch size of 1. This is especially significant because the static method can take hundreds or thousands of iterations to converge in some cases, so our algorithm would provide large savings of resources in these applications. Finally, we see a greater speedup in both time and iterations for the smaller batch sizes of 1 and 10. As mentioned earlier, this is because as the batch size increases, the dynamic algorithm nears the work of a static algorithm. This shows that the dynamic approach is most useful for monitoring applications where the rankings must be updated after only a small number of data changes.

Next we examine the behavior of both algorithms with respect to raw iteration counts over time. Henceforth when referring to the static algorithm, we use Method 2 from above. Figure 4.4 plots the raw number of iterations used by the static (the



(a) $b = 1$ (b) $b = 10$

(c) $b = 100$ (d) $b = 1000$

Figure 4.4: Raw number of iterations for the FACEBOOK graph for different batch sizes. Dynamic algorithm is plotted in solid green line and static algorithm is plotted in dotted blue line.

dotted blue line) and dynamic (the solid green line) algorithms for different batch sizes for the FACEBOOK graph. We sample at 100 evenly spaced timepoints for each batch size. Figures 4.4a, 4.4b, 4.4c, and 4.4d plot the comparison for batch sizes $b = 1, 10, 100, 1000$ respectively. All four figures show the same general behavior: while the number of iterations for static recomputation continues to steadily increase as edges are added into the graph, the dynamic algorithm maintains a stable number of iterations over time. This is because the dynamic algorithm only targets the places in the vector that are affected by edge updates. For example, take Figure 4.4b. The dotted blue line shows that the number of iterations for the static recomputation of the centrality vector continually increases over time as more edges are added into the

graph, eventually reaching about 175 iterations once all edges are added. However, for the dynamic algorithm shown in the solid green line, the number of iterations is stable at around 1-20 iterations for all points in time. It is important to note that this trend persists regardless of the batch size. Even for very large batch sizes of $b = 1000$, while there are small fluctuations in the number of iterations, there is no trend of increasing iteration counts over time, meaning our algorithm is robust to many edge insertions.

Table 4.6: Summary statistics of recall of top vertices for different graphs for a terminating tolerance of 10^{-4}.

Type	**Graph**	**Top 10**	**Top 100**	**Top 1000**
a) Our dynamic algorithm				
Global	facebook	1.00	1.00	1.00
	gowalla	1.00	1.00	1.00
	dblp	1.00	0.99	1.00
	dogster	1.00	1.00	1.00
	youtube	1.00	1.00	1.00
Personalized	facebook	1.00	1.00	1.00
	gowalla	1.00	1.00	1.00
	dblp	1.00	1.00	1.00
	dogster	1.00	1.00	1.00
	youtube	1.00	1.00	1.00
b) alternate approach				
Global	facebook	0.91	0.84	0.89
	gowalla	0.92	1.00	0.99
	dblp	1.00	0.93	0.92
	dogster	1.00	0.95	0.96
	youtube	1.00	0.97	0.95
Personalized	facebook	0.89	0.94	0.91
	gowalla	0.90	0.93	0.96
	dblp	0.95	0.97	0.95
	dogster	0.98	0.92	0.91
	youtube	0.93	0.82	0.87

We have seen that we are able to achieve results faster using a dynamic algorithm compared to static recomputation every time the graph changes when calculating centrality scores in dynamic networks. However, it is also important to ensure that the centrality scores returned by the dynamic algorithm are similar to those returned

by the static algorithm. To evaluate the quality of our algorithm, we measure two quantities: 1) recall of top k vertices measured as

$$recall_k = \frac{|C_S(k) \cap C_D(k)|}{|C_S(k)|},$$

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

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

Table 4.6 presents the average recall of the top 10, 100, and 1000 vertices in the different graphs for both our dynamic algorithm and the alternate approach presented in Section 4.1.2. We use a terminating tolerance of 10^{-4}. Immediately we note that our algorithm has a perfect recall of the top k vertices in all cases except for one graph (DBLP) for one value of k=100, and the recall is 0.99 here. The quality of the alternate approach suffers and is not able to maintain perfect recall in many cases. Furthermore, will see next that the actual values of the scores themselves (measured by the average error) between the dynamically computed vector from the alternate method compared to static recomputation are not similar at all, and we obtain very high errors using this alternate method.

Table 4.7 presents the average error for each of the graphs tested and for all batch sizes for both our dynamic method and the alternate method. We again use results from a tolerance of 10^{-4}. The average error obtained from our dynamic algorithm for global and personalized scores is 1.32e-02 and 8.69e-05 respectively. However, the average error obtained from the alternate approach compared to static recomputation for global and personalized scores is 6.19e+03 and 1.14e-01 respectively. For both global and personalized scores, the errors from the alternate method are several orders

Table 4.7: Summary statistics of average error versus batch size for different graphs for a terminating tolerance of 10^{-4}.

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

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

Next we look at the behavior of both the alternate method and our dynamic algorithm over time. Figure 4.5 plots the average error over time for our dynamic algorithm (the figures on the left) and the alternate method (the figures on the right). We show results for a batch size of 1 for global scores, although results for other

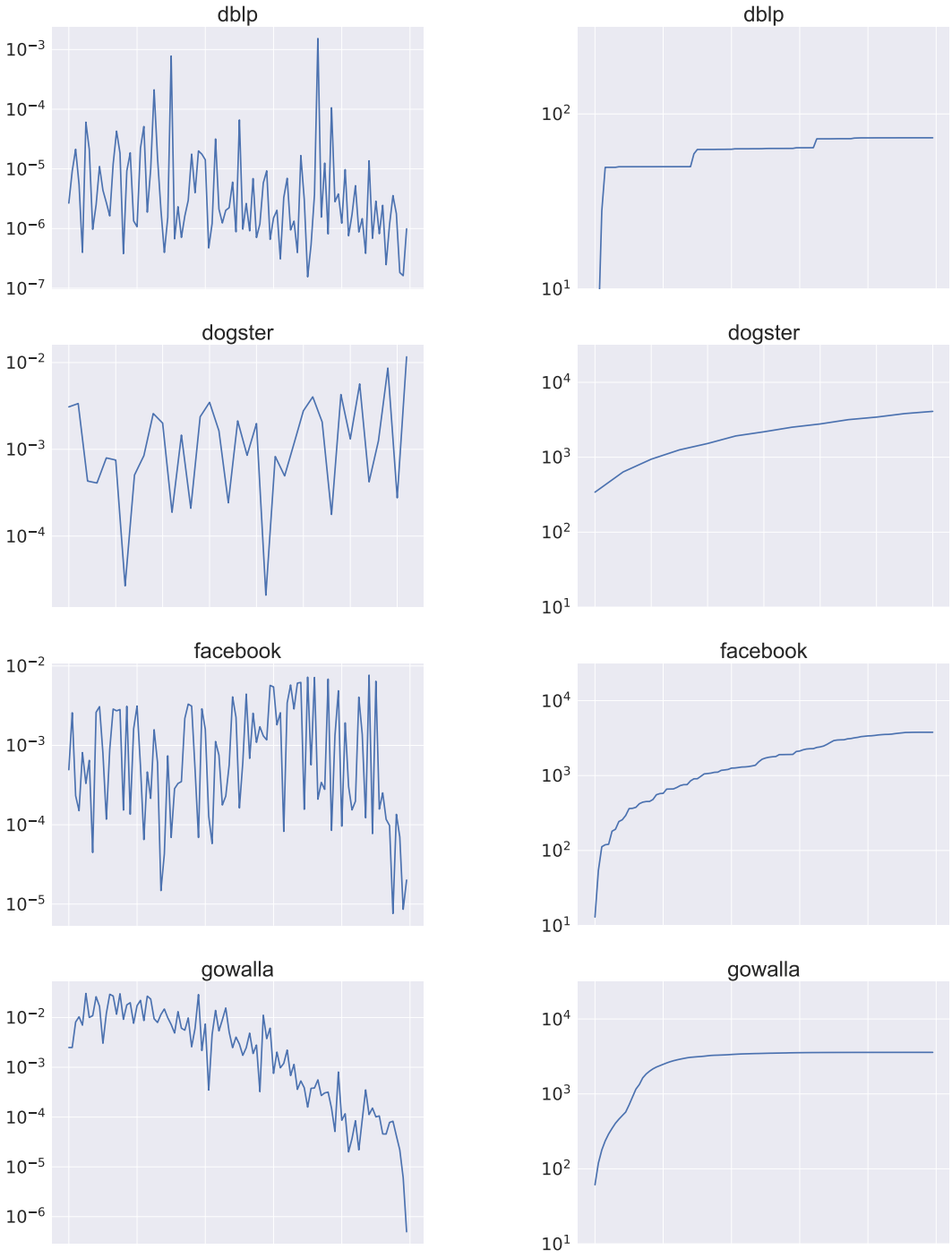


Figure 4.5: *Continued on next page.*

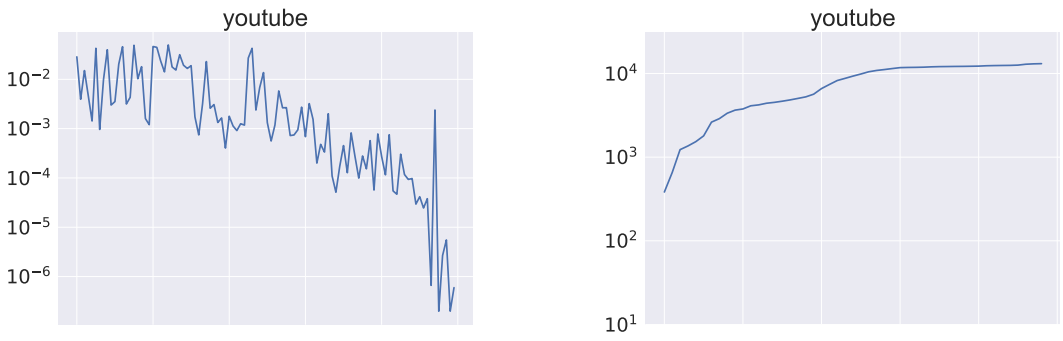
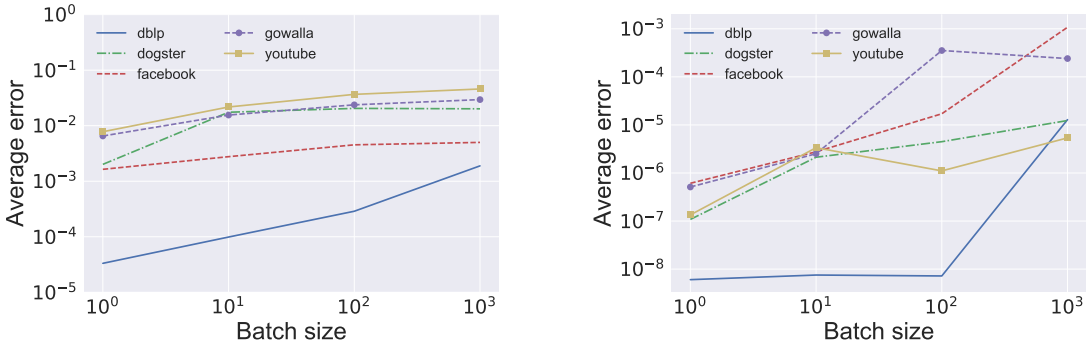


Figure 4.5: Average error plotted over time for both our dynamic algorithm (left figures) and the alternate method (right figures). Results are shown for a batch size of 1 and for global scores. Lower values are better.

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

Finally, Figure 4.6 explores in detail the underlying impact of the time step granularity on the quality of our algorithm. Figure 4.6a plots the error versus batch size for the global scores and Figure 4.6b plots the error versus batch size for the personalized



(a) Average error versus batch size for global scores.

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

Figure 4.6: Effect of time step granularity (batch size of edge insertions) on quality of our algorithm.

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

4.1.5 Adding and Removing Vertices

Adding and removing edges is fairly straightforward since edges only require updating the ΔA matrix with either a 1 (insertions) or -1 (deletions) in the corresponding position for the edge in question. However, adding and removing vertices becomes slightly trickier, since our work is based in linear algebra with fixed size matrices. One solution to this is to assume some reasonable bound on the total number of

vertices allowed (this can be application dependent or based on available storage). The algorithm would then start with a matrix A_0 with empty rows for vertices that do not exist yet in the graph and as the vertices are added with edges into the existing graph, the corresponding rows are also updated. Deleting a vertex can be handled in a similar manner by allowing the vertex to technically exist but remain disconnected from the entire graph. Essentially when deleting vertex i from the graph, we can cope by zeroing out the ith row in the adjacency matrix.

4.1.6 Conclusions

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

negligible. Moreover, our algorithm returns perfect recall of top vertices across all graphs in nearly all cases.

4.2 Agglomerative Personalized Katz Centrality

In this section, we present a new algorithm for approximating personalized Katz Centrality scores in static graphs (STATIC_KATZ) and extend our algorithm for dynamic graphs (DYNAMIC_KATZ). We show STATIC_KATZ provides good quality approximations for personalized scores and is several orders of magnitude faster in time when compared to the conventional linear algebraic method of computing personalized Katz scores. DYNAMIC_KATZ is faster when compared to a pure static recomputation and preserves the ranking of vertices in evolving networks. We present results on both synthetic and real-world graphs. We present our algorithms in Section 4.2.2. Section 4.2.3 evaluates our methods with respect to performance and quality, and in Section 4.2.5 we conclude.

4.2.1 Background

We first recall some relevant background to motivate our work. As previously discussed, Katz Centrality scores ($\mathbf{c}$) count the number of weighted walks in a graph starting at vertex i, penalizing longer walks with a user-chosen parameter α. A walk of length k in a graph traverses edges between a series of vertices $v_1, v_2, \cdots, v_k$, where vertices and edges are allowed to repeat. Powers of the adjacency matrix allow us to count walks of different lengths between vertices in the graph, where $A^k(i, j)$ gives the number of walks of length k from vertex i to vertex j. To count weighted walks of different lengths in the graph, we can sum powers of the adjacency matrix using the infinite series

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

Provided α is chosen to be within the appropriate range ($|\alpha| < \|A\|_2$), this infinite series converges to the matrix resolvent $(I - \alpha A)^{-1}$. Here we concern oursevles with the personalized Katz scores with respect to vertex i, calculated as $(I - \alpha A)^{-1}\mathbf{e}_i$, where $\mathbf{e}_i$ is the ith canonical basis vector. We set $\alpha = 0.85/\|A\|_2$ as in [21]. Note that this is the same definition of Katz Centrality given in Section 2.3.1, but offset by a constant factor. The rankings remain the same. We study this version of the equation in this section for ease of computation.

Typically Katz Centrality scores are calculated using linear algebra by solving the linear system $\mathbf{c} = (I - \alpha A)^{-1}\mathbf{1}$ for the global scores or $\mathbf{c} = (I - \alpha A)^{-1}\mathbf{e}_i$ [81]. While solving the linear system works fairly well for the global scores, in the personalized case many of the vertices have scores close to 0 if they are very far away from the seed vertex i. Therefore, solving the linear system above for personalized scores becomes increasingly computationally intensive because it requires many iterations to converge. For this reason, in this section we present an agglomerative algorithm as an alternate method to the typical linear algebra approach to calculating approximate personalized Katz scores. We calculate scores by examining the actual network structure itself to count walks without using linear algebra. Our algorithm assumes a single seed vertex but can be extended to allow for multiple seed vertices. Henceforth, we use *seed* to denote the seed vertex (so we are computing personalized Katz scores with respect to vertex *seed*).

4.2.2 Algorithms

First we present our static algorithm, STATIC_KATZ. Since walks in graphs allow for repeats of vertices and edges, calcuating exact Katz Centrality scores involves counting walks up until infinite lengths. In practice this is not feasible and so the algorithm we present calculates only approximate Katz Centrality scores. To approximate scores, we count walks only up to length k. We denote the vector of personalized

Katz scores obtained by only counting walks up until length k w.r.t. *seed* as $\mathbf{c}_k = (I + \alpha A + \alpha^2 A^2 + \cdots + \alpha^k A^k)\mathbf{e}_{seed}$.

The algorithm we present is an iterative one, where at iteration j we count walks of length j. In STATIC_KATZ, we maintain three separate data structures:

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

The overarching static algorithm is given in Algorithm 11 and is split into two subroutines. The first subroutine in Algorithm 12, COMPUTE_WALKS, counts the number of walks. To do so, we implement a variant of breadth-first search. The queue *map* is initialized with the source vertex *seed*. At each iteration j, we perform the following main steps:

1. Iterate through all vertices v in *map* (line 7)
2. If we haven't already visited vertex v, we set the value of $visited[v]$ to the current iteration j (line 9)
3. This is the key step in calculating the number of walks. Here, $N(v)$ indicates the set of neighbors of vertex v. For each neighbor vertex, we propagate the number of walks from v. If there are *count* number of walks from *seed* to v of

length $j-1$, then for each neighbor *dest* of v, there are *count* number of walks from *seed* to *dest* of length j going through v (line 11)

4. Finally, we set the values in the *walks* array for the current iteration j to indicate how many total number of walks are possible from *seed* to all vertices reachable in the current iteration (line 13)

The second subroutine in Algorithm 13, CALCULATE_SCORES, actually calculates the personalized Katz scores using the *walks* array. The Katz score for vertex i is the weighted (by powers of α) sum of walks of all lengths up to k from *seed* to i.

Algorithm 11 Static algorithm to compute Katz scores from source vertex *seed* up to walks of length k.

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

Algorithm 12 Static algorithm to recompute counts of walks up to length k from source vertex *seed*.

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

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

Algorithm 13 Calculate Katz scores from walk counts.

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

exact solution $\mathbf{c}^*$by ϵ_k as follows:

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

Note that this proof means that the scores provided from our approximation will never be greater than ϵ_k away from the exact scores neglecting round-off errors. We will see in Section 4.2.3 that this bound not only provides reasonable results but our approximation empirically produces scores also several orders of magnitude closer than what is theoretically guaranteed and ranking quality is preserved.

While results in Section 4.2.3 only examine problems where we start at a single seed vertex, our algorithm can easily be adapted to the case where we allow multiple seed vertices. Instead of initializing the *map* with only the single seed vertex in Line 4 in Algorithm 12, we simply initialize the map with all desired seed vertices. The rest of the algorithm can remain the same as we will then count walks from all seed vertices. The complexity of our static algorithm is $\mathcal{O}(d_{max}k)$, where d_{max} is the

maximum degree of a vertex in the graph. This is because at each iteration we can touch at most d_{max} edges and we run our algorithm a total of k times to count walks up to length k.

Next we present our dynamic algorithm. The overall dynamic algorithm DYNAMIC_KATZ for updating personalized Katz scores is given Algorithm 14 and uses a helper function UPDATE_WALKS, given in Algorithm 15. For our dynamic algorithm we consider the case where we insert a single edge e into the graph between vertices *src* and *dest*. Instead of a complete static recomputation, we can avoid unnecessary computation by using the previously described *visited* array. If we insert an edge between vertices *src* and *dest*, we only need to update counts of walks for vertices that have been visited after vertices *src* and *dest*. Furthermore, we only need to update counts for walks that use the newly added edge. Given a starting vertex *curr_vtx* and integer j, the function UPDATE_WALKS propagates the updated counts of walks from *curr_vtx* to the remaining vertices starting at walks of length j. We do this by maintaining a queue of walk counts for each vertex visited using a variant of breadth-first search, similar to the static algorithm described earlier. The key step is in line 8, where we only traverse walks and update the walk count if we are using the newly added edge. This effectively prunes the amount of work done compared to a pure static recomputation.

In Algorithm 14, DYNAMIC_KATZ, for an inserted edge e=($src, dest$) we calculate which vertex has been visited first (lines 2-6). Without loss of generality, suppose *src* had originally been visited first. In line 7, we update the *visited* value of *dest* because we can now get to *dest* from *src* using the newly added edge. Accordingly, we increment the number of walks possible for *dest* by one as a direct result of the new edge in line 8. For the inserted edge e, the function DYNAMIC_KATZ calls the helper function UPDATE_WALKS for both affected vertices *src* and *dest* to update the walk counts. For vertex *src*, we start updating walks of length *visited*[*src*]+1 and

similarly for vertex *dest* for walks of length *visited*[*dest*]+1. Adding these updated counts to the existing array *walks* effectively propagates the effect of adding the new edge and then in line 11 we calculate the updated Katz scores. Once we have the updated walks, we can calculate the scores using Algorithm 13 as we did in the static recomputation.

Note that our dynamic algorithm is an approximation to the static recomputation. While updating the walk counts for *src* and *dest* using the new edge accounts for much of the effect of the added edge, it is possible there are walks originating from other vertices in the network that go through the added edge that need to be updated. However, the effect of these extra walks will be minimal compared to the effect from the *src* and *dest* vertices, and we show that our dynamic algorithm maintains good quality compared to a static recomputation when concerned about recall of the highly ranked vertices in Section 4.2.3. The worst-case complexity of our dynamic algorithm is still the same as the static algorithm, $\mathcal{O}(d_{max}m)$, because in the worst-case we may still have to touch d_{max} edges at each iteration. However empirically we see that we still obtain significant speedups compared to the static algorithm in Section 4.2.3 because in practice our dynamic algorithm only traverses an edge if the walk in question uses the newly added edge.

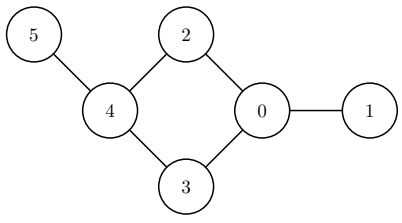
Algorithm 14 Update Katz scores using dynamic algorithm given edge update *edge* from vertex *src* to *dest*

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

Algorithm 15 Helper function for dynamic algorithm to update walks

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

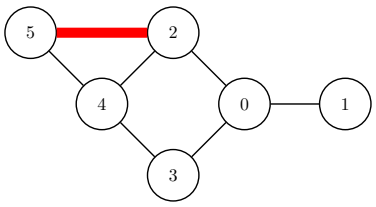
We illustrate our dynamic algorithm on a small toy network. Figure 4.7 depicts the initial graph and the corresponding walk counts of length k up until $k = 3$ for $seed = 0$. In Figure 4.8, we add an edge between vertices 2 and 5 and show the updated walk counts desired in red. The *visited* array is updated accordingly, since we can now reach vertex 5 through vertex 2. When we update the walk counts from vertex 5 starting at walks of length $visited[5]+1 = 3$, we obtain a new walk of length 3 to vertex 2 that uses the new edge $(0 \to 2 \to 5 \to 2)$. When we update the walk counts from vertex 2, we obtain a new walk of length 3 to vertex 4 using the new edge $(0 \to 2 \to 5 \to 4)$.



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

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

Figure 4.7: Initial graph with walk counts of length k and visited values.



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

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

Figure 4.8: Updated graph with walk counts of length k and visited values.

4.2.3 Results

We evaluate StaticKatz and DynamicKatz on synthetic and real-world graphs. For synthetic networks, we use Erdos-Renyi graphs (ER) [79] and R-MAT graphs [80]. In the Erdos-Renyi model, all edges have the same probability for existing in the graph. R-MAT graphs are scale-free networks designed to simulate real-world graphs. For real-world networks, we use four networks from the KONECT collection [76]. Graph information is given in Table 4.8. For all results, five vertices from each graph are chosen randomly as seed vertices and results shown are averaged over these five seeds. Finally, many real graphs are small-world networks [82], meaning the graph diameter is on the order of $\mathcal{O}(\log(n))$, where n is again the number of vertices in the graph. Our algorithm therefore sets $k = \lceil \log(n) \rceil$, so by counting walks up to length $\approx \log(n)$, we can touch most vertices in the graph. The code was implemented in C.

4.2.4 Static Results

For StaticKatz, we present comparisons to the conventional linear algebraic method of computing Katz scores of solving the linear sytem $(I - \alpha A)^{-1}\mathbf{e}_i$. Recall we denote the exact solution given by linear algebra as $\mathbf{c}^*$ and $\mathbf{c}_k$ to represent the personalized Katz scores from StaticKatz. Figure 4.9 plots the absolute error from our algorithm between $\mathbf{c}^*$ and $\mathbf{c}_k$ in the dotted blue line while the theoretically guaranteed error ϵ_k is plotted in the solid green line, where $error = \|\mathbf{c}^* - \mathbf{c}_k\|_2$. Both errors are

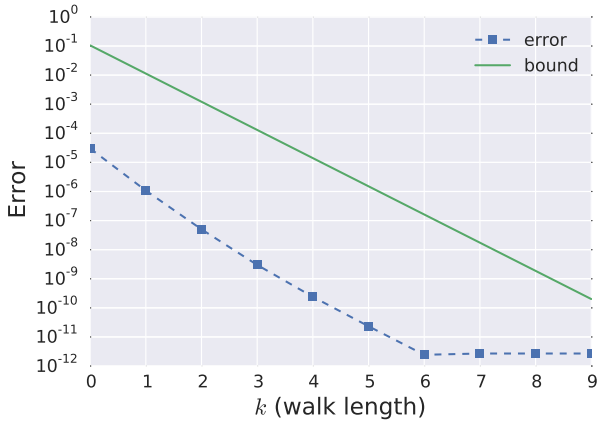


Figure 4.9: Error between approximate scores $\mathbf{c}_k$ and exact solution $\mathbf{c}^*$.

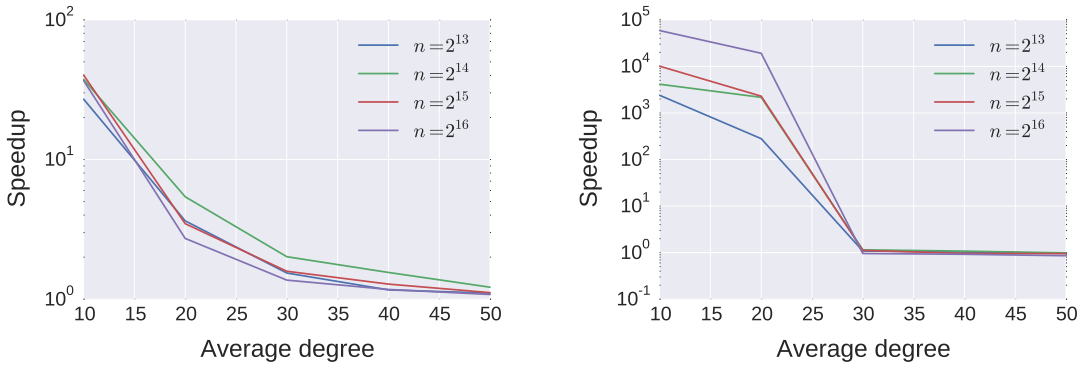
plotted as a function of k. Results are shown only for the MANUFACTURING graph, although similar trends are seen for the other graphs. We see that the actual experimental error is always several orders of magnitude below the theoretically guaranteed error, meaning our algorithm performs better than expected.

In Table 4.8 we summarize the relative speedup obtained from counting walks versus calculating the exact scores using linear algebra for all the real-world graphs by giving the raw times taken by both methods. Let T_L denote the time taken by the linear algebraic method and T_S the time taken by STATIC_KATZ. We note that counting walks using our method is several orders of magnitude faster than linear algebraically computing personalized Katz scores.

Table 4.8: Speedup for real-world networks used in experiments.

Graph	$\boldsymbol{\lvert V\rvert}$	$\boldsymbol{\lvert E\rvert}$	$\boldsymbol{T_L}$	$\boldsymbol{T_S}$
manufacturing	167	82,927	0.74s	0.0059s
facebook	42,390	876,993	132.96s	0.0947s
slashdot	51,083	140,778	241.21s	0.058s
digg	279,630	1,731,653	62.58s	0.053s

We test our method of updating Katz Centrality scores in dynamic graphs on the synthetic ER and R-MAT graphs and on the three largest real-world networks from Table 4.8. Dynamic results are given as comparisons to a pure static recomputation



(a) Erdos-Renyi graphs.

(b) R-MAT graphs.

Figure 4.10: Speedup vs average degree for synthetic graphs tested.

(comparing the performance and quality of DYNAMIC_KATZ to STATIC_KATZ). To have a baseline for comparison, every time we update the centrality scores using DYNAMIC_KATZ, we recompute the centrality vector statically using STATIC_KATZ. Denote the vector of scores obtained by static recomputation as $\mathbf{c}_S$ and the scores obtained by the dynamic algorithm as $\mathbf{c}_D$. We create an initial graph G_0 using the first half of edges, which provides a starting point for both the dynamic and static algorithms. To simulate a stream of edges in a dynamic graph, we insert the remaining edges sequentially and apply both STATIC_KATZ and DYNAMIC_KATZ.

For both ER and R-MAT graphs, we generate graphs with the number of vertices n as a power of 2, ranging from 2^{13} to 2^{16}. We vary the average degree of the graphs from 10 to 50. Denote the time taken by static recomputation and our dynamic algorithm as T_S and T_D respectively. We calculate speedup as T_S/T_D. Figure 4.10 shows the average speedup obtained over time versus the average degree in the graph. For both types of graphs we see the greatest speedup for sparser graphs (smaller average degree). For R-MAT graphs, we also observe greater speedups overall for larger graphs (larger values of n).

For real graphs, we evaluate our algorithm on the three largest graphs from Table 4.8. Let $S_S(R)$ and $S_D(R)$ be the sets of top R highly ranked vertices produced by

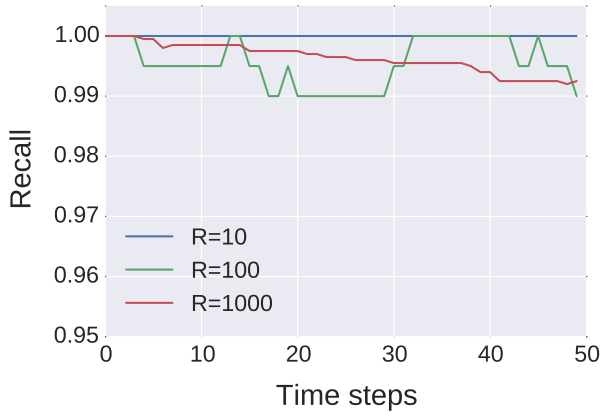


Figure 4.11: Ranking accuracy over time for top R=10,100,1000 vertices for the SLASHDOT graph.

static recomputation and our dynamic algorithm respectively. We evaluate the quality of our algorithm based on two metrics: 1) error = $\|\mathbf{c}_S - \mathbf{c}_D\|_2$, and 2) recall of the top R vertices = $\frac{|S_S(R) \cap S_D(R)|}{R}$. We want low values of the error, meaning DYNAMIC_KATZ produces Katz scores similar to that of STATIC_KATZ, and values of recall close to 1, meaning DYNAMIC_KATZ identifies the same highly ranked vertices as STATIC_KATZ. We consider values of R = 10, 100, and 1000. For many application purposes it is primarily the highly-ranked vertices that are of interest [83]. For example, these may be the most influential voices in a Twitter network, or sites of disease origin in a network modeling disease spread. Showing that our algorithm maintains good recall on the highly ranked vertices has many practical applications.

Table 4.9 gives averages over time of the performance and quality of our algorithm. For the three graphs tested, our dynamic algorithm is several thousand times faster than static recomputation. Average recall of the top R vertices is very high in all cases (greater than 0.99), showing that our approximation of Katz scores is accurate enough in dynamic graphs to preserve the top highly ranked vertices in the graph. The values of the error, although relatively small, indicate that our dynamic algorithm does not find exactly the same scores as a static recomputation. Therefore, our

Table 4.9: Averages over time for real-world graphs for dynamic algorithm compared to static recomputation. Columns are graph name, speedup, absolute error, and recall for $R = 10$, 100 and 1000.

Graph	**Speedup**	**Average Recall**			**Error**
		$\boldsymbol{R = 10}$	$\boldsymbol{R = 100}$	$\boldsymbol{R = 1000}$	
facebook	27,674.50×	1.00	0.997	0.999	0.081
slashdot	47,278.82×	1.00	0.995	0.996	0.013
digg	60,073.81×	1.00	0.996	0.991	0.037

dynamic algorithm should be used if a user's primary purpose is recall of highly ranked vertices without concern of the exact values of the scores.

Furthermore, we observe that the quality of our algorithm does not suffer over time and is therefore robust to many edge insertions. Figure 4.11 plots the recall over time (sampled at 50 evenly spaced timepoints) for the SLASHDOT graph for the top $R = 10$, 100 and 1000 vertices. Note that the y-axis starts at 0.95. We are able to maintain a high recall of the top ranked vertices with little to no decrease over time. The results for other graphs tested are similar.

4.2.5 Conclusions

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

Table 4.10: Several walk-based centralities as functions of the adjacency matrix

Centrality Metric	**Generalized Equation**
Katz Centrality PageRank	$\sum_{k=0}^{\infty} \alpha^k A^k$
Eigenvector centrality	$A\mathbf{x} = \lambda\mathbf{x}$
Exponential centrality Subgraph centrality Total communicability	$\sum_{k=0}^{\infty} \frac{A^k}{k!}$

4.3 Nonbacktracking Walk Centrality

This section presents a new dynamic algorithm for calculating updated centrality values using nonbacktracking walks. Section 4.3.1 motivates the study of centrality using nonbacktracking walks and Section 4.3.2 describes the new algorithms. Section 4.3.3 presents the main results of our method and in Section 4.3.4 we conclude.

4.3.1 Background & Motivation

We start this section by again reviewing the definition of a walk in a graph. Denote a walk of length k as a series of vertices $v_1, v_2, \cdots, v_k$, where vertices and edges are allowed to repeat. Using linear algebraic notation, we can count walks of different lengths using powers of the adjacency matrix A where $A^k(i, j)$ gives the number of walks of length k from vertex i to j. As discussed earlier, several centrality metrics are calculated as functions of the adjacency matrix and weight walks of different lengths to quantify importance. A subset of these centrality metrics and their generalized equation is given in Table 4.10.

The similarity amongst all these walk-based centrality metrics stems from the fact that they weight all walks of the same length equally. For example, a walk of length 4 between vertices $0 \to 1 \to 0 \to 1 \to 0$ is weighted the same as a walk of length 4 between vertices $0 \to 1 \to 2 \to 3 \to 4$. The authors in [84] propose a new measure of centrality based on the concept of a *nonbacktracking walk* (NBTW), a walk which does

not backtrack upon itself, meaning it contains no vertex sequences of the form iji. Thus, nonbacktracking walk centrality scores are computed by counting NBTWs in graphs and weighting longer ones by successive powers of some parameter $\alpha \in (0, 1)$. Specifically in this section we again study personalized centrality (w.r.t. seed vertices of interest), but using NBTWs instead of regular walks as for Katz Centrality. We count NBTWs originating at some seed vertex and ending at all other vertices in the graph.

When NBTW-based centrality was first introduced in [84], the authors presented a linear algebraic formulation for calculation of the centrality scores. Solving the linear system in Equation 4.3 for an $n \times 1$ vector $\mathbf{x}^*$ gives the centrality scores. The vector $\mathbf{e}_i$ indicates we are solving for personalized scores w.r.t. a seed vertex i and Δ is the associated diagonal degree matrix of the adjacency matrix A.

$$(I - \alpha A + \alpha^2(\Delta - I))\mathbf{x}^* = (1 - \alpha^2)\mathbf{e}_i \tag{4.3}$$

However, for large graphs, this linear system is computationally intensive to solve and for personalized scores, the scores of vertices far away from the seed are often negligible. Therefore, it is desirable to have an alternate method to calculate these centrality scores and in this work we present one such alternate algorithm. Since walks (and NBTWs) in graphs can be infinitely long, if we are counting walks manually (without using linear algebra), we can approximate the corresponding centrality metric by counting walks up to a certain length.

4.3.2 Algorithms

This section first presents an algorithm for approximating NBTW-based centrality in static graphs, which serves as a starting point for the dynamic algorithm. Suppose we are counting walks originating at a seed vertex *seed*. For a graph with n

vertices, we maintain an $n \times k$ array *walks* where *walks[i][j]* represents the number of nonbacktracking walks from *seed* to vertex i of length j. Let $N(i)$ denote the set of neighbors of i. For a particular NBTW w that ends with the sequence of vertices $\cdots, j, i$, let $\widetilde{N}(i) = N(i) \backslash j$, meaning the set of neighbors of vertex i without the vertex the NBTW w came from (vertex j). The effect of a walk from *seed* to one of its direct neighbors can be propagated recursively throughout the network, where we only advance the walk to a vertex if we don't backtrack. Since walks are required to be nonbacktracking, at step k we need to keep track of the vertex that was visited at step $k-1$. We can think of propagating a walk through the network as examining the neighbors of the last vertex visited in the walk and updating walk counts of its neighbors as long as we don't backtrack. Specifically, for each neighbor vertex $d \in \widetilde{N}(v)$, there will be $walks[v][k]$ walks of length $k+1$ ending at d going through v.

The main computation in counting the NBTWs occurs in Algorithm 16, where we propagate the effect of a NBTW ending at a particular vertex throughout the entire network. Suppose we have already calculated a NBTW of length k ending in the sequence of vertices $\cdots, prev, curr$. We examine the vertex *curr* and look at $\widetilde{N}(curr)$, or equivalently the set of *curr*'s neighbors that don't include *prev*, since these are the set of vertices that our NBTW can visit next (Line 4). For each vertex *vtx* in this set we update the number of NBTWs of length $k+1$ that now end in the sequence $\cdots, prev, curr, vtx$ in Line 8. We now recursively repeat this calculation replacing *curr* with *vtx* (Line 9).

Algorithm 16 can be used to develop an algorithm for counting all NBTWs up to length k_{max} starting at *seed*. This procedure is given in Algorithm 17. By the definition of a NBTW, the only vertices that will have a NBTW of length 1 from *seed* are the seed vertex's direct neighbors. Thus, Line 2 first obtains the neighbors of *seed*. For each neighbor vertex *nbr* we will propagate the effect of the walk from *seed* to *nbr* throughout the rest of the network using the previously described PROPAGATE

Algorithm 16 Propagate num_walks walks from $seed$ to $curr$ of length $k+1$ going through $prev$.

```
1: procedure PROPAGATE(prev, curr, k, num_walks, walks, k_max)
2:     if k > k_max then
3:         return
4:     S = N(curr)\prev                          ▷ Can't backtrack through prev
5:     if S is ∅ then                ▷ If ∃ 0 neighbors to propagate walks, return
6:         return
7:     for vtx ∈ S do
8:         walks[vtx][k + 1] += num_walks
9:         PROPAGATE(seed, vtx, k + 1, num_walks, walks, k_max)
       return walks
```

function in Algorithm 16 in Line 6.

Algorithm 17 Static algorithm to calculate personalized NBTW centrality.

```
1: procedure STATIC_NBTW(seed, k_max)
2:     Nbrs = N(seed)
3:     for nbr ∈ Nbrs do
4:         num_walks = 1
5:         walks[nbr][1] = num_walks              ▷ ∃ 1 walk from seed to nbr
6:         PROPAGATE(seed, nbr, 1, num_walks, walks, k_max)
       return walks
```

We prove the correctness of our algorithm STATIC_NBTW in Theorem 7.

Observation 1. If there are x NBTWs from $seed$ to v of length k, then there are x NBTWs of length $k+1$ from $seed$ to each of the vertices in $\widetilde{N}(v)$ going through v. Alternately, the number of NBTWs of length $k+1$ from $seed$ to vertex i can be calculated by summing up the number of NBTWs of length k from $seed$ to $\widetilde{N}(i)$. Note that this follows trivially from the definition of a NBTW.

Theorem 7. *For all vertices i, Algorithm 17 updates $walks[i,k]$ with the correct number of NBTWs of length k from seed to vertex i.*

Proof. We will prove this by induction.

Base case. $k = 1$: The only NBTWs that exist in the network from $seed$ of length 1 are its direct neighbors. This is taken care of in the initialization of the algorithm

in Line 4.

Inductive hypothesis. Assume $walks[i, k]$ holds the correct number of NBTWS of length k from *seed* to i for all vertices i.

Inductive step. We will show that it follows that $walks[i, k+1]$ holds the correct number of NBTWS of length k from *seed* to i.

- First note that NBTWs ending in i must first traverse through a neighbor of i before reaching i, i.e. for a NBTW w of length $k+1$ to end in i, the first k vertices of w must be a NBTW of length k ending in some $v \in N(i)$. Furthermore, in order to have a NBTW, we cannot backtrack so we can impose the additional constraint of $v \in \widetilde{N}(i)$.

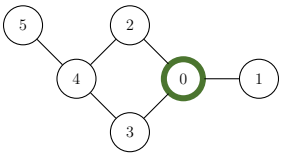
- By the inductive hypothesis, note that $walks[v, k]$ holds the correct number of NBTWs of length k from *seed* to vertex v $\forall v$. By Observation 1, $\forall v \in \widetilde{N}(i)$, $walks[i, k+1] += walks[v, k]$.

- Therefore, $walks[i, k+1]$ correctly counts the number of NBTWs from *seed* to i of length $k+1$.

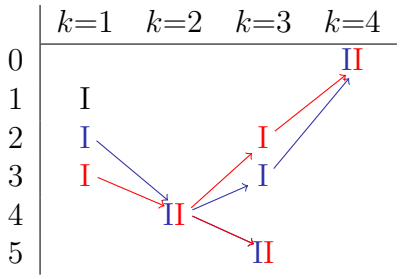
□

Figure 4.12 gives an example of our static algorithm on a toy network. The example graph is shown in Figure 4.12a with a seed vertex 0 outlined in green. Figure 4.12b shows how our algorithm propagates walks from the seed vertex 0. Vertex 0 has three direct neighbors, vertices 1, 2, and 3. Since the NBTW from $0 \to 1$ can't backtrack onto 0 (vertex 1's only neighbor), this NBTW ends here. However, the NBTWs $0 \to 2$ and $0 \to 3$ are propagated through the network as shown in the *walks* array, where the result of the NBTW $0 \to 2$ is shown in blue and the result of the NBTW $0 \to 3$ is shown in red.

For dynamic graphs, a naive implementation to obtain updated NBTW counts after changes to the graph occur would recompute from scratch the number of NBTWs



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



(b) Walks array.

Figure 4.12: Example of STATIC_NBTW. Propagation of different walks is shown in different colors. For a seed vertex of 0, we propagate walks from neighbors vertex 1, 2, and 3 throughout the network.

from *seed*. However, as the graph grows larger, this naive static recomputation becomes increasingly computationally intensive. By exploiting the locality of edge insertions we can develop a more efficient dynamic algorithm that only updates NBTW counts relevant to new edges inserted into the graph. We consider the case of inserting a single edge $e = (src, dest)$. Our dynamic algorithm is given in DYNAMIC_NBTW in Algorithm 19. We will repeat the same set of steps to obtain updated NBTW counts for both the *src* and *dest* vertices. This set of steps is given in DYNAMIC_HELPER in Algorithm 18. Without loss of generality, let us first consider the effect of the *src* vertex. All current NBTWs ending in *src* need to be updated since we can now visit *dest* from *src* by traversing the newly added edge. To identify which NBTWs need to be looked at, we first find all the values of k where $walks[src][k]$ is nonzero in Line 2 (obtained in the array k_vals). If $walks[src][k] > 0$, then there are a nonzero number of NBTWs of length k that end in *src* and we need to propagate these using the newly added edge e. Line 3 obtains the numbers of walks of length j for each $j \in k\_vals$ in the array num_walks_vals. The same technique of propagating walks can be used as described earlier in Line 7. The same procedure described for the *src* vertex can then be applied to the *dest* vertex. Lines 8-11 takes care of the edge case when either *src* or *dest* is the seed vertex. In this case we need to perform a full propagation from the start similar to the static algorithm. Since we only propagate

walks that use the newly added edge, we save on computation time had we performed a full recomputation.

Algorithm 18 Helper function for dynamic update.

```
1: procedure DYNAMIC_HELPER(src, dest, seed, walks)
2:     k_vals = walks[src].nonzero
3:     num_walks_vals = [walks[src][k_vals[i]]]
4:     for i in len(k_vals) do
5:         k = k_vals[i]; num_walks = num_walks_vals[i]
6:         walks[dest][k + 1]+ = num_walks
7:         PROPAGATE(src, dest, k + 2, num_walks, walks, k_max)
8:     if src is seed then                ▷ Edge case if new edge uses seed vertex
9:         num_walks = 1
10:        walks[dest][1] = num_walks
11:        PROPAGATE(seed, dest, 1, num_walks, walks, k_max)
```

Algorithm 19 Dynamic algorithm to calculate personalized NBTW centrality given new edge e.

```
1: procedure DYNAMIC_NBTW(seed, k_max, edge = (src, dest), walks)
2:     DYNAMIC_HELPER(src, dest, seed, walks)
3:     DYNAMIC_HELPER(dest, src, seed, walks)
4: return walks
```

We prove the correctness of our algorithm DYNAMIC_NBTW in Theorem 8. Assume we start with the NBTW counts at time t from our static algorithm in the array *walks*. We seek to prove that upon addition of a single edge $e = (src, dest)$ at time $t + 1$, our dynamic algorithm produces the same result as would have been obtained from a complete static recomputation using the previously described static algorithm.

Theorem 8. *Upon addition of a single new edge $e = (src, dest)$, for all vertices i, Algorithm 19 updates walks$[i, k]$ with the correct number of NBTWs of length k from seed to vertex i, where correctness here is measured as computing the same NBTW counts as static recomputation.*

Proof. We will prove this by induction. Let *walks_static* be the NBTW counts produced from static recomputation and *walks_dynamic* be the NBTW counts produced

by our dynamic algorithm.

Base case. $k = 1$: NBTWs of length 1 can only be affected by the new edge if *src* or *dest* is the seed vertex. This is taken care of in line 8 in Algorithm 18.

Inductive hypothesis. Assume $walks\_dynamic[i, k] = walks\_static[i, k]$ for all vertices i.

Inductive step. We will show that it follows that $walks\_dynamic[i, k + 1]$ $= walks\_static[i, k + 1]$.

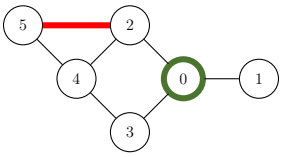
- Observe that the only new NBTWs that have to be accounted for are those using the new edge e. Specifically, NBTWs ending at *src* can now travel to *dest* and vice versa.

- Suppose by the inductive hypothesis that we have x NBTWs of length k from *seed* to *src*. By the previous point, we have x new NBTWs of length $k + 1$ from *seed* to *dest* (i.e., all NBTWs ending at *src* of length k can now travel to *dest* creating NBTWs of length $k + 1$). This is similar to the logic the static algorithm employs.

- This logic is implemented in Line 6 in Algorithm 18. Since we account for all new NBTWs, $walks\_dynamic[i, k + 1] = walks\_static[i, k + 1]$.

□

Since we are not recalculating all the counts of NBTWs for all the vertices from *seed*, and are only examining the effect of a single edge and the effect it has, this dynamic approach will be significantly faster than a naive static recomputation every time the graph is changed and we see this in Section 4.3.3. Figure 4.13 gives an example of our dynamic algorithm using the same toy network as earlier. Consider the effect of adding a single edge $e = (2, 5)$ (shown in red in Figure 4.13a). Figure 4.13b gives the initial NBTW counts for the network before adding edge e. After



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

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

(b) Initial walks array before adding edge e.

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

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

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

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

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

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

Figure 4.13: Example of DYNAMIC_NBTW. After adding edge e between vertices 2 and 5 we show the steps of the dynamic algorithm to update NBTW counts taking into consideration the new edge.

inserting the edge between vertices 2 and 5, there are three NBTWs and their counts to update: 1) the NBTW of length 1 starting at vertex 2, 2) the NBTW of length 3 starting at vertex 2, and 3) two NBTWs of length 3 starting at vertex 5. These propagations are given in Figures 4.13c, 4.13d, and 4.13e respectively.

Both STATIC_NBTW and DYNAMIC_NBTW return an $n \times k$ array *walks* that can then be used to calculate the centrality scores. This procedure is given in Algorithm 20 where we obtain the centrality value for vertex i by weighting NBTWs of different lengths by successive powers of some user-chosen parameter α.

Algorithm 20 Calculate NBTW-centrality scores from walk counts.

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

4.3.3 Results

We evaluate STATIC_NBTW and DYNAMIC_NBTW on five real-world graphs drawn from the KONECT collection [76]. Graph information is given in Table 4.11. For all results, five vertices from each graph are chosen randomly as seed vertices and results shown are averaged over these five seeds. We use temporal datasets to simulate dynamic graphs, meaning the edges already have associated timestamps. For our dynamic algorithm, we initialize the algorithm with half the edges and then insert the remaining edges in different batch sizes in timestamped order. We test batch sizes of 1, 10, 100, and 1000. A batch size of b means at each time point we insert b edges and run both the dynamic and static algorithms for comparison purposes. As previously discussed, many real graphs are small-world networks [82], meaning the graph diameter is on the order of $\mathcal{O}(\log(n))$, where n is again the number of vertices in the graph. Our algorithm therefore sets $k = \lceil \log(n) \rceil$, so by counting walks up to length $\approx \log(n)$, we can touch most vertices in the graph. The code was implemented in C++.

Table 4.11: Real graphs used in experiments.

Graph	$\|V\|$	$\|E\|$
wiki-news	25,042	193,618
facebook	46,952	876,993
wiki-talk	534,767	2,271,361
wiki-french	1,409,666	3,853,639
youtube	3,223,585	9,375,374

For our static algorithm we present comparisons to a conventional linear algebraic method of solving the system in Equation 4.3 discussed in Section 4.3.1. The goal here is to ensure our algorithm returns similar quality scores to a traditional linear algebraic computation of centrality scores. Let $\mathbf{x}^*$ be the solution to the linear system (the exact NBTW-centrality scores) and $\mathbf{x}_k$ be the approximation from our algorithm by counting up to length k NBTWs. We measure error as the 2-norm difference

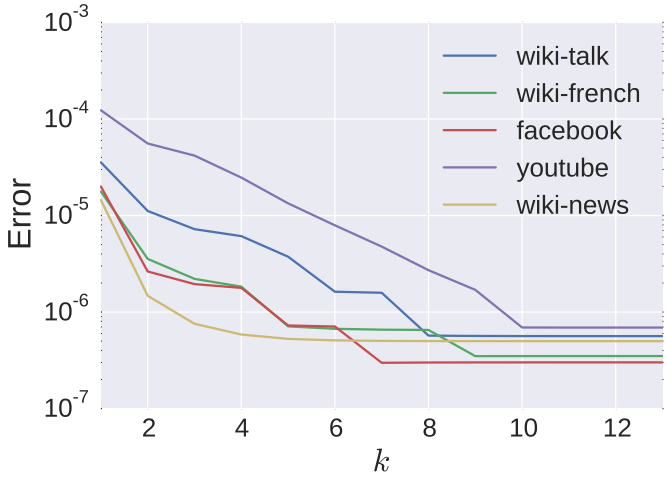


Figure 4.14: Absolute error between exact NBTW-centrality scores $\mathbf{x}^*$ and our approximation $\mathbf{x}_k$.

between the two vectors as

$$error = \|\mathbf{x}^* - \mathbf{x}_k\|_2.$$

Figure 4.14 plots the error (on the y-axis) for different values of k (on the x-axis) for all the real graphs. The first trend to note is the most intuitive: as we include counts of longer lengths in the calculations of the scores, the error between our approximation and the exact scores decreases. Note that after a certain value of k, the error stabilizes and we can conclude that counting further walks of longer lengths has no significant impact on the quality of the scores. Therefore, setting the value of k that we count to to some constant is a viable choice in our methods.

Our dynamic algorithm produces the same NBTW counts as our static algorithm (and therefore, the same scores), so we only examine the performance of our dynamic algorithm w.r.t. speedup compared to the static algorithm. Let T_S be the time taken by our static algorithm to compute the NBTW-based centrality scores for a particular graph and T_D be the time taken by our dynamic algorithm. To evaluate our dynamic

algorithm, we calculate the speedup in time as

$$speedup = \frac{T_S}{T_D}.$$

Higher values of the speedup indicate our dynamic algorithm has significant performance improvement compared to our static.

Figure 4.15 plots the maximum, mean, and minimum speedup over all the real graphs (on the y-axis) versus the batch size (on the x-axis). In most cases even the minimum speedup obtained is above $1\times$ and very rarely does it drop below a $1\times$ speedup. We see the greatest speedup for smaller batch sizes of 1 and 10, indicating that our method is most beneficial for low latency applications with small number of data changes. The average speedup obtained decreases for larger batch sizes. This is due to the fact that as the batch size grows larger, the amount of time needed to process the updates grows because all endpoints of all edges newly added must be taken into account. Essentially, new NBTWs must be propagated from all the touched endpoints of the newly added edges. However, our dynamic algorithm still on average is able to obtain several orders of magnitude in speedup over the static recomputation. In very large graphs of billions of vertices where a static recomputation is computationally infeasible given edge updates to a graph, our dynamic algorithm offers significant savings because it just targets a localized portion of the graph where the edge has been added.

Figure 4.16 plots the speedup over time for each of the different graphs tested. We sample at 100 evenly spaced time points and plot the time taken (in seconds) for our dynamic (in the solid blue line) and our static algorithm (in the dotted green line). We see that the time taken by our dynamic algorithm is several orders of magnitude lower than the time taken by our static algorithm. This indicates that our method of only examining places in the graph that are directly affected by the edge updates

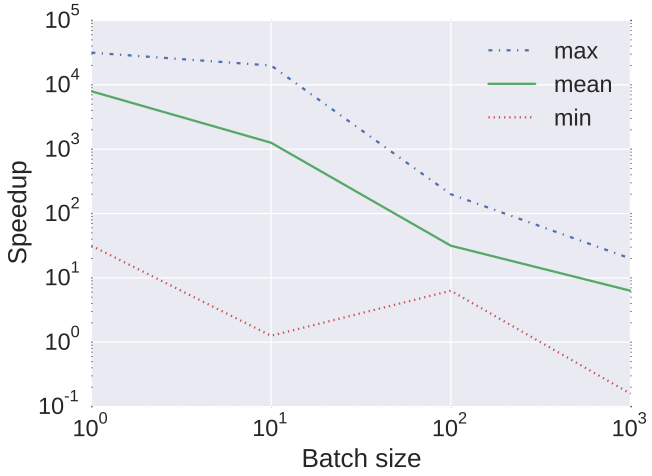
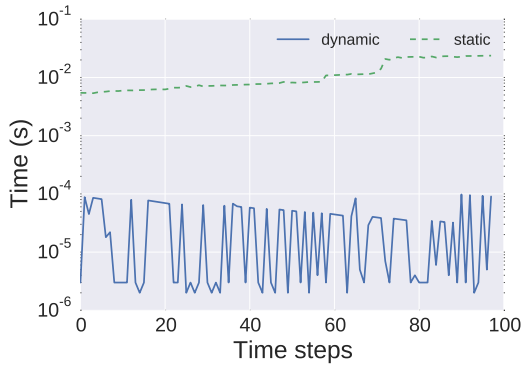


Figure 4.15: Speedup versus batch size for real graphs. Higher is better.

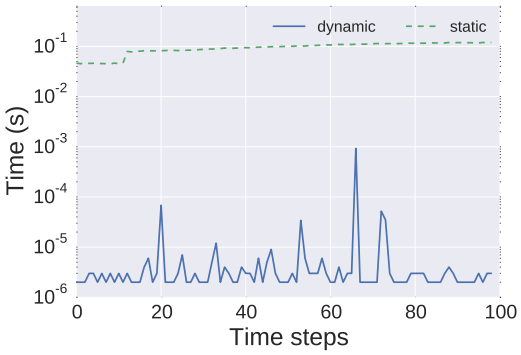
results in highly efficient computation of NBTW-based centrality scores.

4.3.4 Conclusions

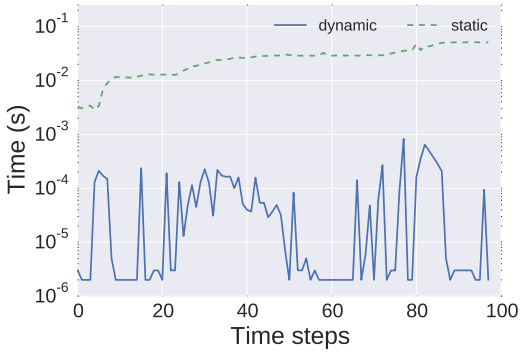
This section presented a new algorithm for computing the values of personalized non-backtracking walk-based centrality scores of the vertices in both static and dynamic graphs. The algorithm returns approximations of scores by counting NBTWs up to a certain length starting at a given seed vertex. In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



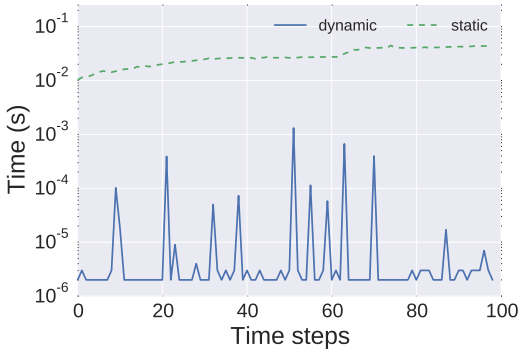
(a) WIKI-NEWS graph.



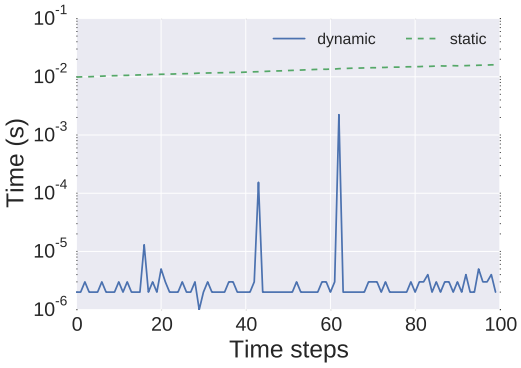
(b) FACEBOOK graph.



(c) WIKI-TALK graph.



(d) WIKI-FRENCH graph.



(e) YOUTUBE graph.

Figure 4.16: Speedup in time of dynamic algorithm compared to static algorithm for real graphs.

the computation in both the static and dynamic algorithm; however this is out of the scope of this dissertation. For example, in the propagation step, if a vertex has three neighbors, the propagation of those walks are independent from each other and can be done in parallel. However, care would need to be taken to ensure that the recursive nature of propagating walks does not spawn too much parallelization, which could cause too much overhead and negate any performance benefit obtained.

4.4 Streaming Exponential Centrality

This section presents a new dynamic algorithm for updating exponential-based centrality scores in evolving graphs. Our method is faster than standard static recomputation and maintains high recall of the highly ranked vertices over time. Section 4.4.2 presents our new dynamic algorithm and Section 4.4.3 presents experimental results on both synthetic and real-world graphs.

4.4.1 Background

Recall that subgraph centrality is determined by the diagonal elements of some matrix function applied to the adjacency matrix A of the graph under study [20]. A frequent function of choice is the matrix exponential e^A [85]. Consider the power series expansion of e^A [86]:

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots + \frac{A^k}{k!} + \cdots = \sum_{k=0}^{\infty} \frac{A^k}{k!},$$

where I is the $n \times n$ identity matrix. Since $A^k(i,j)$ counts the number of walks of length k between vertices i and j, the diagonal elements of e^A ($e^A(i,i)$) count the number of closed walks (starting and ending at the same vertex) centered at vertex i weighting a walk of length k by $\frac{1}{k!}$. Here, we use the diagonal elements of the matrix exponential, $e^A(i,i)$, as the centrality scores for the vertices. An alternate means of

calculating centrality scores from the matrix exponential is to use the row sums of e^A, since in practice this is faster than obtaining the diagonal elements, which requires computation of the entire matrix. However, various results in previous literature have shown that these two methods (row sums versus only the diagonal elements) often produce fairly different rankings and so we cannot simply replace one with the other [21]. Since our analysis of exponential centrality requires calculating the entire matrix, which is a dense matrix, this work focuses on medium sized graphs; however future work can consist of using methods to approximate the matrix exponential to scale to larger graphs.

4.4.2 Methodology

The goal of our dynamic algorithm is to prune unnecessary computation when calculating the updated centrality scores of the vertices in the graph after edge updates occur. Therefore, our algorithm uses the computations from the previous timestep in the calculation of the scores in the current timestep. This forms the basis of our dynamic algorithm. We obtain updated snapshots of the adjacency matrix at time $t+1$ as $A_{t+1} = A_t + \Delta A$, where ΔA represents the edge udpates occurring at time $t+1$.

The end goal is to calculate $e^{A_{t+1}}$, or equivalently $e^{A_t+\Delta A}$. Since we are working with exponentials, a naive first pass algorithm is to attempt to exploit basic properties of exponentials, namely the additive property. However, the additive property of exponentials fails for matrices unless we have commutativity: for $n \times n$ matrices $A = B + C$, $e^A \neq e^B e^C$ unless $BC = CB$. Since we cannot trust graph updates to be commutative, this naive additive property alone is not sufficient for our purposes and we cannot simply compute $e^{A_{t+1}}$ as $e^{A_t} e^{\Delta A}$. However, there is still a relationship between the parts of the sum for the matrix exponential as stated in Theorem 9 that we can use to develop a streaming algorithm for the matrix exponential in dynamic

graphs.

Theorem 9. *Suppose $A = B + C$ where A, B, and C are $n \times n$ matrices. Then the exponential of A is related to the exponentials of B and C by the Trotter product formula [87]:*

$$e^A = \lim_{m \to \infty} (e^{B/m} e^{C/m})^m.$$

Furthermore, the Trotter result can be used to approximate e^A by using the approximation [88]:

$$e^A \approx (e^{B/m} e^{C/m})^m.$$

Suppose we have the matrix exponential of A at time t, e^{A_t}. Given edge updates ΔA to the graph, our goal is to compute the updated matrix exponential $e^{A_{t+1}}$ with minimal computation. Using Theorem 9, we can calculate $e^{A_{t+1}}$ as:

$$\begin{aligned} e^{A_{t+1}} &= e^{A_t + \Delta A} \\ &\approx (e^{A_t/m} e^{\Delta A/m})^m \end{aligned}$$

As the value of m increases, although we obtain better quality approximations, the computation time increases. Since there is an inverse relationship between quality and performance, in this work we use values of $m = 2$ and 3 and results shown are averaged between these two parameter values. We see in practice that we obtain high quality results from this setting.

4.4.3 Results

We test our algorithm on both synthetic and real-world graphs. For synthetic networks, we test two types: preferential attachment and small-world. The preferential attachment graphs are built using the Barabási-Albert model [89] and possess a scale-free degree distribution. The graph is created by adding vertices one by one. The

model takes two parameters: n and d, where n is the number of vertices in the graph and d is the number of edges each new vertex is given when it is first inserted into the graph. To create a scale-free distribution, edges of the newly inserted vertex connect to vertices already in the network with a probability proportional to the degree of the existing vertices. Small-world networks are build using the Watts-Strogatz model [90]. This model produces graphs with high levels of clustering as seen in real networks and with small graph diameter (the small-world property). This model takes three parameters: n, d, and p, where n is the number of vertices in the graph, which are arranged in a ring and connected to their d nearest neighbors. Each vertex is then independently considered and with probability p an edge is placed between the vertex and a randomly chosen vertex. Here, we fix p at 0.1. For both types of graphs (Barabási-Albert and Watts-Strogatz) we use values of $n = 1000$, 2000, and 3000 and vary d from 1-10. For real graphs, we draw from the KONECT [76] collection of datasets, listed in Table 5.3. All the real graphs are temporal networks, meaning the edges have timestamps associated with them.

For the experiments, edges are permuted randomly for synthetic networks and inserted in timestamped order for the real graphs. To simulate a dynamic graph, we insert edges in batch sizes of 2^i for $i = 0, 1, 2, 3, 4, 5, 7$, and 9. Specifically, at each timepoint t, 2^i more edges are added to the graph. We compare the performance and quality of our dynamic algorithm to the standard static algorithm of recomputing the matrix exponential from scratch every time the underlying graph is changed. The code was implemented in Python and we use SciPy's built in expm function to calculate the matrix exponential.

For synthetic graphs, we show results for a batch size of 1. First we measure performance of our dynamic algorithm. Let T_S denote the time taken by the static recomputation averaged over all points in time and let T_D denote the time taken by

Table 4.12: Real graphs used in experiments.

Graph	$\|\boldsymbol{V}\|$	$\|\boldsymbol{E}\|$
facebook	2,888	2,981
power-grid	4,941	6,594
ca-HepTh	9,877	25,998
wb-cs-stanford	9,435	36,854

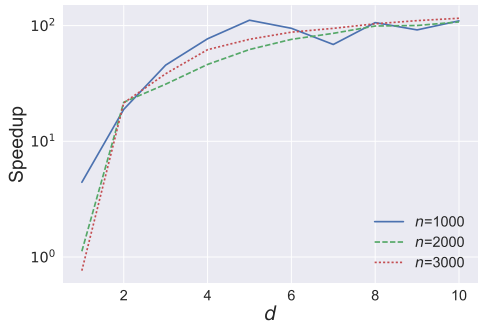
our dynamic algorithm. To measure performance, we calculate speedup as

$$speedup = \frac{T_S}{T_D}.$$

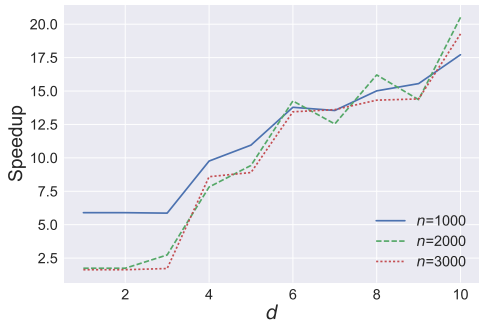
Values greater than 1 indicate that our dynamic algorithm is faster than a pure static recomputation. Figures 4.17a and 4.17b plot speedup versus d for the preferential attachment graphs and small-world graphs, respectively. The speedups for the preferential attachment graphs are several orders of magnitude higher than the corresponding small-world networks. For both types of graphs however, as the graph becomes denser (larger values of d), the speedups increase. The speedups seen can be attributed to two factors: the sparsity of ΔA and the rate of convergence of $e^{\Delta A}$ versus that of e^{A_t}. Since ΔA only consists of the edge updates at a particular time point, this matrix contains far fewer entries than that of A_t and therefore the calculations needed for the matrix exponential for ΔA will converge far quicker than those needed for the full matrix A_t as is required by the static algorithm.

Next we evaluate the quality of our dynamic algorithm with respect to static recomputation. Many applications in data analysis are concerned with only the highly ranked vertices in graphs [91]. Therefore to measure quality, we calculate recall of the top R vertices for $R = 25$, 50, and 100. Let $C_S(R)$ be the set of the top R highly ranked vertices from static recomputation and $C_D(R)$ be the set of the top R vertices from our dynamic algorithm. Then recall of the top k vertices is calculated as

$$recall_R = \frac{|C_S(R) \cap C_D(R)|}{R}.$$



(a) Speedup for preferential attachment graphs.



(b) Speedup for small world graphs.

Figure 4.17: Speedup for synthetic graphs for batch size $2^0 = 1$.

Values close to 1 indicate that our algorithm identifies a high percentage of the highly ranked vertices compared to the solution from static recomputation. Tables 4.13 and 4.14 show values of the recall for the top 25, 50, and 100 highly ranked vertices for different values of d for the preferential attachment and small-world graphs, respectively. For both types of graphs we average over $n = 1000$, 2000 and 3000. We observe that the recall values for the preferential attachment graphs are higher than their small world counterparts. This can be attributed to the different degree distributions of the two types of graphs. Due to the manner of creation of the small-world networks, the topology of the network is relatively homogeneous and all vertices have essentially the same degree. In contrast, the preferential attachment graphs have hubs and a scale-free degree distribution. The difference in rankings of vertices is more likely much more prominent in graphs with a scale-free degree distribution (the preferential attachment graphs) compared to graphs with a much more homogenous degree distribution (the small-world graphs). In graphs where all vertices have a similar degree it is likely that the centrality scores themselves are also fairly similar. Since our dynamic algorithm is an approximation to the statically recomputed scores, with similar centrality scores, the rankings can themselves be easily interchanged for similarly valued vertices. Therefore it is not surprising that the recall values for the

Table 4.13: Recall values for preferential attachment graphs.

d	recall_{25}	recall_{50}	recall_{100}
1	0.88	0.88	0.88
2	0.90	0.91	0.91
3	0.88	0.88	0.89
4	0.87	0.87	0.88
5	0.85	0.85	0.86
6	0.84	0.84	0.85
7	0.82	0.83	0.83
8	0.80	0.80	0.81
9	0.77	0.77	0.79
10	0.75	0.76	0.76

Table 4.14: Recall values for small world graphs.

d	recall_{25}	recall_{50}	recall_{100}
1	0.77	0.78	0.80
2	0.77	0.78	0.80
3	0.77	0.78	0.82
4	0.78	0.80	0.83
5	0.80	0.82	0.83
6	0.77	0.80	0.84
7	0.72	0.73	0.80
8	0.73	0.78	0.80
9	0.71	0.76	0.80
10	0.68	0.72	0.78

small-world graphs are lower than their preferential attachment counterparts. Furthermore, while the recall values for the preferential attachment graphs decrease as values of d increase, there is no such trend for the small world graphs, which tend to have fairly constant values of recall for different values of d.

Note again that these results are averaged over values of $m = 2$ and 3. As mentioned earlier, there is an inverse relationship between computational cost and quality of our algorithm with respect to choosing the paramater m. Specifically, as we increase the value of m, we would obtain recall values approaching closer to 1, but at a higher computational cost.

Next we evaluate our dynamic algorithm on the real graphs from Table 5.3. In terms of performance, Figure 4.18 plots the speedup versus batch size (note that both axes are on a log scale base 2 for clarity). We are able to obtain up to a 32× speedup for batch sizes larger than $2^3 = 8$ with a median speedup of about 16×, and we always have greater than a 1× speedup. As the batch size increases, the speedup obtained increases to a certain point, after which it plateaus at an average of around 16× speedup.

Next we examine the quality of our algorithm on real-world graphs. Table 4.15 gives the average recall values over all points in time for all batch sizes for all graphs

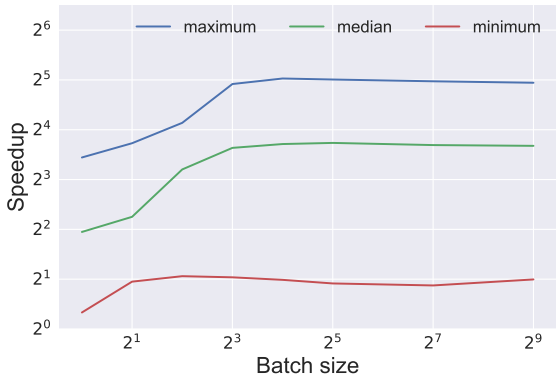


Figure 4.18: Speedup versus batch size for real graphs.

for the top R highly ranked vertices for $R = 25$, 50, and 100, and gives the average over all batch sizes. In most cases, the average recall is over 0.75 indicating our algorithm is able to retrieve a large percentage of the highly ranked vertices compared to static recomputation. There is also a slight trend of increasing values of recall with larger batch sizes, though the average recalls over all batch sizes are fairly high. Figure 4.19 plots the recall over time for all the graphs for a batch size of 128, though trends for other batch sizes are similar. The x-axis simulates time as we insert more edges into the graph and the y-axis plots the recall at that point in time. The most important trend we note is that while there are occasionally dips in the recall values over time, there is no overall trend of the quality worsening over time. This indicates that at no point in time is there evidence that we need to restart our dynamic algorithm. In fact, for some of the graphs (WB-CS-STANFORD and CA-HEPTH) the recall actually increases over time.

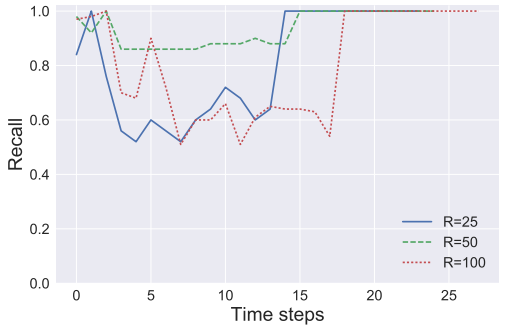
Finally in addition to recall, we examine the Kendall rank correlation coefficient (τ) , a measure of the correspondence between two rankings [92]. For two $n \times 1$ vectors $\mathbf{x}$ and $\mathbf{y}$, we define P to be the number of concordant pairs (the number of elements where the ranks given by both $\mathbf{x}$ and $\mathbf{y}$ agree) and Q to be the number of discordant pairs. For example, a pair of elements (i, j) is *concordant* if both $x_i > x_j$

Table 4.15: Recall for real-world graphs.

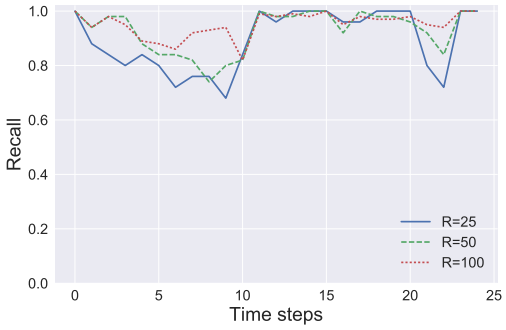
Graph	Top R	Batch size								Average
		2^0	2^1	2^2	2^3	2^4	2^5	2^7	2^9	
facebook	R=25	0.61	0.59	0.79	0.79	0.72	0.80	0.74	0.73	0.72
	R=50	0.83	0.81	0.93	0.93	0.93	0.93	0.89	0.98	0.90
	R=100	0.61	0.61	0.80	0.78	0.77	0.80	0.76	0.76	0.74
power-grid	R=25	0.83	0.86	0.86	0.86	0.87	0.89	0.89	0.92	0.87
	R=50	0.84	0.87	0.87	0.86	0.86	0.88	0.93	0.87	0.87
	R=100	0.86	0.88	0.87	0.88	0.88	0.90	0.95	0.92	0.89
wb-cs-stanford	R=25	0.52	0.66	0.80	0.90	0.95	0.96	0.96	0.97	0.84
	R=50	0.58	0.56	0.70	0.89	0.94	0.94	0.93	0.94	0.81
	R=100	0.69	0.66	0.69	0.84	0.90	0.90	0.89	0.92	0.81
ca-HepTh	R=25	0.68	0.63	0.79	0.86	0.88	0.89	0.83	0.76	0.79
	R=50	0.57	0.63	0.73	0.82	0.83	0.81	0.80	0.73	0.74
	R=100	0.60	0.72	0.77	0.84	0.86	0.86	0.85	0.79	0.79

Table 4.16: Values of τ for real graphs.

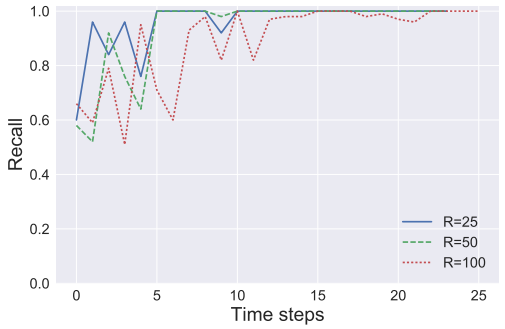
Graph	τ
facebook	0.747
power-grid	0.812
ca-HepTh	0.528
wb-cs-stanford	0.761



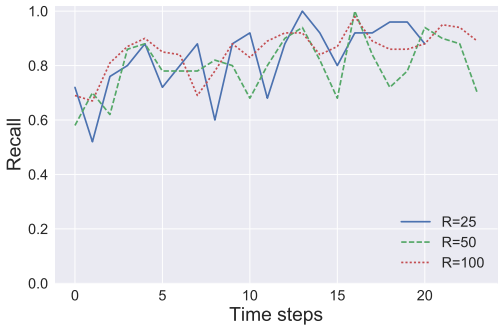
(a) FACEBOOK graph.



(b) POWER-GRID graph.



(c) WB-CS-STANFORD graph.



(d) CA-HEPTH graph.

Figure 4.19: Recall over time for different graphs for batch size $2^7 = 128$.

and $y_i > y_j$ or if both $x_i < x_j$ and $y_i < y_j$. They are *discordant* if $x_i > x_j$ and $y_i < y_j$ or if $x_i < x_j$ and $y_i > y_j$. The Kendall τ coefficient is calculated as

$$\tau = \frac{P - Q}{n(n-1)/2}.$$

We compare the rankings given by the entire statically recomputed vector versus the vector obtained from our dynamic algorithm. Values close to 1 indicate strong agreement whereas values close to -1 indicate strong disagreement. Specifically, if the two rankings agree perfectly (they provide the same rankings for all pairs of vertices) we expect a value of 1. Similarly, if the two rankings disagree perfectly, τ would be -1. A value of 0 indicates the two rankings have no relationship to each other. Table 4.16 gives the values of τ for the real graphs averaged over all batch sizes and over

all points in time. We note that for all real graphs tested, the value of τ is above 0 indicating that there is always agreement between the statically computed vector and dynamically computed vector. For all but one of the real graphs, the value of τ is above 0.7, indicating a strong agreement in the rankings of all the vertices.

4.4.4 Conclusions

In this section, we presented a new algorithm for computing the values of exponential-based centrality in dynamic graphs by studying the matrix exponential. We tested our method on both synthetic and real-world graphs and observe that our dynamic algorithm outperforms static recomputation. Additionally, the quality of our method is robust and does not decay over time, meaning that since there is no significant drift, there is no evidence that we would need to recompute the values at any point in time. Since this work compares the quality of our streaming algorithm to the exact computation of the matrix exponential (which is a computationally heavy task), the graphs used were fairly small. However, future work can consist of scaling our algorithm to larger graphs, which would include investigation of alternative methods of approximating the matrix exponential. Additionally, future work can compare rankings obtained from using the diagonal entries of the matrix exponential to row sums and observing how these rankings change over time in dynamic graphs. While promising avenues of research, these are out of the scope of this dissertation.

CHAPTER 5
LOCAL COMMUNITY DETECTION IN DYNAMIC GRAPHS

This chapter extends previous work done in [6, 17] by applying it to the problem of local community detection, defined further in Section 5.1.2. Previous work on updating Katz Centrality and PageRank are necessary steps towards tracking "relevant" subgraphs around seed vertices using personalized centrality metrics. Specifically, the main contribution of this chapter is to tie together the two fields of community detection and centrality by studying how personalized centrality metrics can be used for local community detection in not only static but also dynamic graphs. We present a new method of identifying local communities using personalized centrality metrics. Section 5.2 presents comparisons to a modified version of greedy seed set expansion, the most commonly used method of finding local communities in graphs. Results show high recall values comparing our method to ground truth on stochastic block model graphs and several orders of magnitude of speedup obtained using our method. Next we present a dynamic algorithm to identify local communities in evolving networks. We see recalls of over 0.80 for synthetic networks showing community evolution and speedups of over 60× execution time improvement compared to static recomputation for real graphs. Our dynamic method returns good quality communities measured by conductance and normalized edge cut and the quality of communities is preserved over time for real graphs. Comparisons using multiple seeds for our algorithm show our method is robust to using many seeds. We review relevant literature regarding community detection and centrality metrics in Section 5.1. Section 5.2's preliminary results include initial validation of our method on static graphs. The algorithms for application to dynamic graphs as well as a thorough discussion of experiments and results appear in Section 5.3. Finally, we conclude and discuss future research

directions in Section 5.4.

5.1 Community Detection in Graphs

5.1.1 Measures of Community Quality

Since there is no universal definition of a community, there are several metrics that exist to evaluate the quality of a community. Several of these metrics focus on calculating how tightly knit a community is in terms of comparing the number of inter-community edges to the number of intra-community edges. Let k_{in}^{C} denote the number of intra-community edges for community C; that is, the number of edges (i, j) with both endpoints vertex i and j inside the community. Similarly, let k_{out}^{C} denote the number of inter-community edges, or the number of edges (i, j) where vertex i is in the community and vertex j is outside the community. Conductance (ϕ) is a popular measure for measuring the "fitness" of a community by measuring the community cut, or the inter-community edges. Conductance is calculated as

$$\phi(C) = \frac{k_{out}^{C}}{min(2k_{in}^{C} + k_{out}^{C}, 2k_{in}^{VC} + k_{out}^{VC})}$$

in [53]. When optimizing a community with respect to conductance, we seek to minimize conductance scores. A lower conductance score indicates a more tightly knit community. Another popular metric for evaluating the quality of communities is to calculate a modified ratio of intra- to inter-community edges, or a normalized edge cut (f) [60] as

$$f(C) = \frac{2k_{in}^{C} + 1}{2k_{in}^{C} + k_{out}^{C}}.$$

Here, a larger value of the normalized edge cut indicates a more tightly knit community, so methods that optimize for the value of the normalized edge cut of a community seek to maximize $f(C)$. Modularity (Q) compares the number of intra-community

edges to the expected number under a random null model and is calculated as

$$Q(C) = k_{in}^{C} - \frac{(2k_{in}^{C} + k_{out}^{C})^2}{4|E|}$$

in [93]. Again, larger values of modularity indicate higher quality communities so algorithms optimizing for modularity seek to maximize values of modularity. Several other metrics were used in recent DIMACS (Center for Discrete Mathematics and Theoretical Computer Science) challenges: the intra-cluster density is defined as $\frac{k_{in}}{\binom{|C|}{2}}$ and coverage of a community is calculated as $\frac{k_{in}}{|E|}$. For a more detailed list of metrics to measure community quality, see [94].

5.1.2 Community Detection

The main contribution of this chapter is to present a new algorithm for local community detection in graphs, specifically how to use a centrality vector indicating relative importance in vertices to identify local communities for seed vertices in a dynamic graph.

Clauset presented a greedy algorithm that starts with all seed vertices in the community and repeatedly checks all neighboring vertices for inclusion [95]. At each iteration, the neighboring vertex that most increases the chosen fitness score is added to the community. This method is shown in Algorithm 21, for a given graph G and seed set of vertices *seed*. Here, C represents the community and $N(C)$ is the set of vertices neighboring C, or those with an edge to a vertex in C. In order to grow a community of k vertices, not including any seeds, it is necessary to perform k iterations and in each iteration check each neighboring vertex. Therefore, the complexity depends on the number of vertices bordering the community. This number may be approximated by kd, where d is the average degree of the graph and k the community size. In this case, the time complexity is given by $\mathcal{O}(k^2 d)$. However, this is an overes-

timate when community members share many common neighbors, such as in graphs with a high clustering coefficient. In Section 5.2, we use synthetic, static graphs to compare the results of our method to this greedy seed set expansion algorithm. We show that our centrality based approach produces high quality communities compared to a common greedy approach and we explain when our approach is faster and preferable.

Algorithm 21 Static, Greedy Seed Set Expansion

```
1:  procedure GreedySeedset(graph G, seed set seed)
2:      C = seed
3:      progess = True
4:      while progress do
5:          maxscore = −1
6:          maxvtx = null
7:          for v ∈ N(C) do
8:              s(v) = fit(C ∪ v) − fit(C)
9:              if s(v) > maxscore then
10:                 maxscore = s(v)
11:                 maxvtx = v
12:         if maxscore > 0 then
13:             C = C ∪ maxvtx
14:         else
15:             progess = false
16:     return C
```

There has also been some work in relating centrality measures and community detection, though much of the previous work has focused on the global or static case. Betweenness centrality as a measure of vertex importance was originally introduced by Freeman in [30] to measure influence of a vertex over the flow of information between other nodes by counting shortest paths in a network. The works by Girvan and Newman [93, 96] extend the definition of vertex betweenness centrality to define edge betweenness as the number of shortest paths between pairs of vertices that run along it. Assuming communities in graphs are connected by only a few inter-community edges, these edges will have high edge betweenness. Therefore, by iteratively removing edges with the highest edge betweenness, community structure can be uncovered. A

greedy local community algorithm using centrality metrics is the L-shell method [97], in which vertices are added to the community from successive shells, or sets of vertices at a fixed distance from the seed vertex. PageRank-Nibble is a spectral method of finding local communities in which personalized PageRank scores are computed and the community is formed by adding vertices with the highest values [98]. Our work differs from these previous works because we incrementally update scores to perform dynamic local community detection. Section 5.2.2 compares our results with static expansion using Katz Centrality in the place of PageRank.

5.1.3 Centrality for Community Detection

This chapter presents a method for local community detection using personalized centrality using methods presented in [6] and [17] as the base for this work. A similar method can apply to non-backtracking variants of Katz Centrality [99] as well. Personalized PageRank vectors and conductance scores have also been used to identify communities in graphs [100]. This method is based off of the fact that the personalized PageRank vector is the stationary distribution of a random walk that follows an edge of the graph with probability α and "teleports" back to the seed vertex with probability $1 - \alpha$. The PageRank scores are calculated by an algorithm that pushes scores to neighboring vertices at each stage using the algorithm described in [98]. Once the PageRank vector is calculated, the algorithm performs a sweep cut to identify the optimal community. This procedure sweeps over all cuts induced by the ordering of the personalized PageRank vector and chooses the best cut determined by conductance scores of the induced cuts. The entire personalized PageRank matrix, formed with each column starting from the corresponding vertex, has been shown asymptotically to recover the stochastic block model used here as a test case [101].

5.2 Communities from Personalized Centrality

5.2.1 Local Communities from Personalized Centrality

Given a seed set of vertices of interest, we can calculate the personalized Katz or PageRank scores as $\boldsymbol{c}_{Katz} = A(I - \alpha A)^{-1}\boldsymbol{b}_{Katz}$ or $\boldsymbol{c}_{PR} = M_{PR}^{-1}\boldsymbol{b}_{PR}$, respectively, where $M_{Katz} = I - \alpha A$ and $M_{PR} = I - \alpha A^T D^{-1}$ with $\boldsymbol{b}_{Katz}$ and $\boldsymbol{b}_{Katz}$ are the corresponding right-hand sides as discussed in Chapter 3. If we want the personalized scores w.r.t. vertex i, then $\boldsymbol{b}_{Katz} = \boldsymbol{b}_{PR} = \boldsymbol{e}_i$. Intuitively, the resultant scores from a personalized centrality metric with respect to vertex i answers the question of how likely we are to reach vertex i from the rest of the graph. For the question of local community detection, this can be translated into how likely vertices in the graph are to belong to the community of vertex i. For a community of size R, we therefore take the top R vertices as ranked by the personalized centrality vector $\boldsymbol{c}_{PR}$ or $\boldsymbol{c}_{Katz}$ as the local community.

Once personalized Katz Centrality or PageRank scores are computed, the local community is then formed from those vertices with highest centrality values. Sorting the entire length n vector to obtain these top entries is too computationally expensive, especially in the dynamic setting where updated results are needed quickly after changes occur. Therefore, we extract the vertices with top k values using a heap. For the first k vertices, the centrality values are added to a heap. Thereafter, each centrality score is compared to the minimum value in the heap in $\mathcal{O}(1)$ time and if larger, the minimum value is removed from and the new value inserted into the heap in $\mathcal{O}(\log k)$ time. In the worst case, the centrality values are in ascending order and all such checks result in a removal and insertion, leading to a running time of $\mathcal{O}(n \log k)$. However, experiments on real graphs show far fewer replacements.

5.2.2 Results on Static, Synthetic Graphs

This section validates using personalized centrality for local community detection. Static, synthetic graphs with known community structure provide test cases for the Katz Centrality approach. We also compare our approach to the popular method of greedy expansion [95], which was described in Section 5.1.2. To test, we generate multiple stochastic block model (SBM) graphs with varying parameters, randomly choose seed vertices, and detect local communities with both personalized Katz Centrality and greedy expansion from Algorithm 21. The greedy expansion method uses conductance as its fitness function.

A simple stochastic block model graph can be generated with four parameters: the total number of vertices n, the number of communities k, the average degree of vertices d, and the percentage of inter-community edges ρ. All communities in such a graph are generated with the same parameters and are interchangeable. Note that SBM graphs can also be generated with different parameters than the ones listed here (as we show in Section 5.3.2). Instead of using an average degree and proportion of inter-community edges, the parameters p_{in} and p_{out} can be used. These define the probabilities of placing and edge between a pair of vertices that are in the same community and between a pair in different communities, respectively. Although different parameters are used, these two models are the same when all communities are generated with the same parameters. The parameters are related as follows. For a set community size of $c = \frac{n}{k}$, $p_{in} = \frac{d(1-\rho)}{c}$ and $p_{out} = \frac{d\rho}{c(k-1)}$. All code for this chapter was implemented in Python.

Since the SBM graphs are generated manually, we know the exact ground truth. Tables 5.1a–c show the recall of communities found with each method compared to the known ground truth. The recall is the fraction of the ground truth recovered by each method. For a known community of size R, let $C_K(R)$ be the community recovered by our Katz approach and $C^*(R)$ be the ground truth community. We

calculate recall as

$$recall = \frac{|C^*(R) \cap C_K(R)|}{|C^*(R)|}.$$

Recall for the greedy seed set method is calculated similarly compared to the ground truth. For these results, random stochastic block model graphs were created with 1000 vertices and two communities and a random seed was chosen. The minimum, mean, and maximum recall values shown are obtained from 100 runs, each with a random graph and seed vertex. For the results shown in Table 5.1a, the average degree of vertices varies from 5 to 490, while the proportion of inter-community edges is fixed at 0.01. All others are intra-community edges. Because the proportion is fixed, as the average degree increases, both the number of intra-community and inter-community edges increases. Overall, recall scores are at or near 1, showing that the Katz method returns good communities for all average degrees considered.

This suggests that using personalized centrality is a viable method of local community detection. In fact, on SBM graphs with low degrees, the personalized Katz method performs better than greedy expansion. This occurs because the greedy expansion method stops adding new vertices once a local quality maximum is reached. On very low degree SBM graphs, it stops expanding after adding only a few vertices, which results in very small communities and thus low recall. Therefore, we also show results for a modified version of the greedy algorithm in which expansion is forced to occur until the community reaches the desired size (labeled *Force Expand* in Tables 5.1a–c). Normally, the greedy algorithm is not run in this way, but because we know the size of the community ahead of time, we can obtain these results. Note that results for the normal greedy algorithm and the forced expansion version tend to differ only for graphs in which the average degree is low compared to the community size.

Tables 5.1b,c show how the quality of communities detected varies for SBM graphs with an increasing proportion of inter-community edges. For these experiments, the average degree is fixed at 20 (Table 5.1b) and 100 (Table 5.1c), and the proportion of

edges that are inter-community varies from 0.01 to 0.4 (thus the proportion of intra-community edges varies from 0.99 to 0.6). As the percentage of inter-community edges increases, the community structure becomes less defined, making community detection more difficult. As expected, all methods achieve the best recall for graphs with a low proportion of inter-community edges. For graphs with a lower average degree of 20, both the Katz and greedy expansion methods return high quality communities only when a small proportion of edges exist between communities. However, when the average degree is increased to 100, both methods are less sensitive to a large proportion of inter-community edges and achieve higher recall values. Overall, the quality of communities returned by the personalized Katz method is comparable to those returned by greedy expansion. While the mean recall is sometimes lower, the minimum recall tends to be higher, making the results more consistent. Note that both the standard greedy and forced expansion greedy algorithms can return communities with very low recall. This may occur if the standard greedy method stops expanding too early or if either version returns the wrong community. Because the method greedily maximizes conductance, if there is a single seed vertex, the next vertex added is its lowest degree neighbor. If this neighbor belongs to a different community, the algorithm may detect and return the wrong community.

An interesting phenomenon occurs in Table 5.1b for SBM graphs with degree 20 and 0.4 inter-community edges. The minimum recall obtained with greedy expansion increases compared to 0.3 inter-community edges. This reversal in trend occurs because, at 0.4 inter-community edges, the community structure is almost gone and the greedy algorithm returns an almost random set of vertices, including many correct vertices. For graphs with a stronger community structure, on the other hand, the minimum recall corresponded to cases in which the greedy algorithm did return a community, but not the correct one.

Next, we consider the relative running time of the personalized Katz approach

compared to the greedy expansion method. Let T_G be the time taken by greedy seed set and T_K the time taken by our Katz method. We calculate relative speedup as

$$speedup = \frac{T_G}{T_K}.$$

Figure 5.1 plots the ratio running times, where a value of x greater than 1 indicates that the Katz method is x times faster than greedy expansion. For these tests, we also use static, synthetic SBM graphs. As before, graphs are generated with four different parameters: the total number of vertices, the number of communities, the average degree of vertices, and the percentage of inter-community edges. For each experiment, three of these parameters are held constant, while one is varied in order to isolate its effect on the running time. The results shown use the modified version of greedy expansion in which the algorithm is forced to expand to the desired community size. We used this version because for those SBM graphs with a very low average degree compared to community size, the standard greedy algorithm stopped expanding after only a few vertices, leading to small and incorrect communities (see Table 5.1). This is likely an artifact of the synthetic SBM graphs in which vertices have uniformly random degrees. For all plots in Figure 5.1, the proportion of inter-community edges ρ is set to 0.01. Overall, we see that using the personalized Katz approach tends to be faster than running the greedy expansion method.

In Figure 5.1a, speedup is shown for graphs with an increasing number of vertices, while the average degree is fixed at 20 and the number of communities is fixed at 2. With all other parameters held constant, the larger the number of vertices in the graph (and therefore the larger the community detected), the greater the speedup of using our Katz approach compared to greedy expansion. This occurs because the complexity of the greedy approach is approximately $\mathcal{O}(c^2 d)$ for a community size of c and average degree d, while the complexity of the Katz approach is $\mathcal{O}(m)$ for a graph

with m edges.

The advantage of our centrality approach compared to greedy expansion is greatest when the size of the community is large relative to the total number of vertices. This can be seen in Figure 5.1b, where we vary the number of communities, while keeping the size of the graph constant at $47,104$ vertices with an average degree of 20. It is clear that the speedup of the Katz method is greatest for the graphs with a small number of large communities. This occurs because the personalized Katz Centrality computation is global and processes the entire graph, while greedy expansion processes only a local subgraph composed of the community and its one hop neighborhood. If, however, the community to be found is much smaller than the full graph, the greedy expansion method may be preferable.

Finally, we consider how the average vertex degree affects relative running times in Figure 5.1c. The number of vertices is held constant at 1000 with 2 communities. As the average degree increases, the speedup of the Katz method over greedy expansion first increases and then decreases. The increase in speedup occurs because a higher average degree results in larger community neighborhoods and the larger the neighborhood of the community is as it expands, the slower the greedy expansion is. However, once the average degree grows large enough, few or no new vertices are added to the neighborhood and the clustering coefficient of the graph simply increases. These results show that the running time advantage of the personalized Katz method compared to greedy expansion is greatest in graphs in which the community of interest has a large neighborhood and a low clustering coefficient.

Table 5.1: The quality of communities detected with our personalized Katz method and greedy expansion is shown. Test graphs are stochastic block model (SBM) graphs with $n = 1000$ and $k = 2$. (a) The average vertex degree d is varied, while $\rho = 0.01$. (b) The proportion of inter-community edges ρ is varied, while $d = 20$. (c) The proportion of inter-community edges ρ is varied, while $d = 100$.

(a)

Avg. Degree	ρ	Katz Recall			Greedy Recall			Forced Greedy Recall		
		Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
5	0.01	0.688	0.936	0.974	0.004	0.015	0.034	0.024	0.924	1.000
10	0.01	0.920	0.988	0.998	0.002	0.104	1.000	0.002	0.970	1.000
20	0.01	0.974	0.997	1.000	0.002	0.902	1.000	0.002	0.990	1.000
50	0.01	0.994	0.999	1.000	0.002	0.990	1.000	0.002	0.990	1.000
100	0.01	0.990	0.998	1.000	0.002	0.990	1.000	0.002	0.990	1.000
250	0.01	1.000	1.000	1.000	0.002	0.990	1.000	0.002	0.990	1.000
490	0.01	1.000	1.000	1.000	0.002	0.990	1.000	0.002	0.990	1.000

(b)

Avg. Degree	ρ	Katz Recall			Greedy Recall			Forced Greedy Recall		
		Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
20	0.01	0.974	0.997	1.000	0.002	0.902	1.000	0.002	0.990	1.000
20	0.05	0.806	0.944	0.988	0.002	0.852	1.000	0.002	0.960	1.000
20	0.1	0.678	0.833	0.910	0.002	0.773	1.000	0.002	0.869	1.000
20	0.2	0.502	0.638	0.730	0.002	0.603	0.998	0.008	0.833	0.998
20	0.3	0.474	0.551	0.630	0.002	0.505	0.932	0.096	0.655	0.942
20	0.4	0.456	0.508	0.542	0.006	0.354	0.594	0.416	0.521	0.604

(c)

Avg. Degree	ρ	Katz Recall			Greedy Recall			Forced Greedy Recall		
		Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
100	0.01	0.990	0.998	1.000	0.002	0.990	1.000	0.002	0.990	1.000
100	0.05	0.980	0.990	1.000	0.002	0.960	1.000	0.002	0.960	1.000
100	0.1	0.942	0.980	0.992	0.002	0.940	1.000	0.002	0.940	1.000
100	0.2	0.728	0.822	0.908	0.002	0.880	1.000	0.002	0.880	1.000
100	0.3	0.552	0.626	0.700	0.002	0.828	1.000	0.002	0.828	1.000
100	0.4	0.482	0.530	0.576	0.070	0.604	0.936	0.074	0.612	0.944

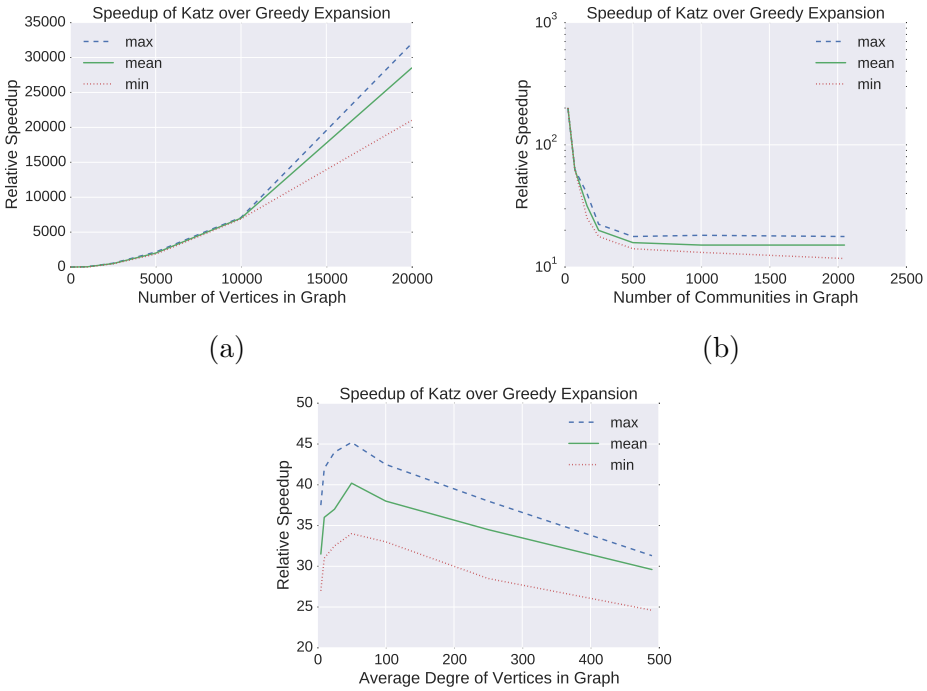


Figure 5.1: The speedup of the personalized Katz Centrality method compared to greedy expansion is shown for SBM graphs with different parameters. (**a**) The number of vertices n in the graph varies, with $d = 20$ and $k = 2$. (**b**) The number of communities k in the graph varies, with $n = 47104$ and $d = 20$. (**c**) The average vertex degree d varies, with $n = 1000$ and $k = 2$.

5.3 Dynamic Communities from Personalized Centrality

5.3.1 Methods

We have detailed how to obtain a local community from a personalized centrality vector in Section 5.2.1. In this section, we describe how to obtain communities in dynamic graphs using personalized centrality metrics. The identification of local communities in dynamic graphs can be split into two components: (1) updating the personalized centrality vector every time the graph changes, and (2) obtaining the new local community from the updated centrality vector.

For both centrality metrics, if $\boldsymbol{c}_t$ denotes the solution at time t, we solve for a correction $\Delta\boldsymbol{c}$ so that we can obtain the new solution at time $t+1$ as $\boldsymbol{c}_{t+1} = \boldsymbol{c}_t + \Delta\boldsymbol{c}$.

Essentially, we use the old solution as a starting point for the new solution instead of recomputing from scratch each time the graph is changed.

Personalized Katz Centrality scores w.r.t. vertex i are given as $\boldsymbol{c}_{Katz} = A\boldsymbol{x}_{Katz}$ where $\boldsymbol{x}_{Katz}$ is the solution to the linear system $(I - \alpha A)\boldsymbol{x}_{Katz} = \boldsymbol{b}_{Katz}$ for $\boldsymbol{b}_{Katz} = \boldsymbol{e}_i$ and personalized PageRank scores are given as $\boldsymbol{c}_{PR} = (I - \alpha A^T D^{-1})^{-1}\boldsymbol{b}_{PR}$, where $\boldsymbol{c}_{PR}$ is the solution to the linear system $(I - \alpha A^T D^{-1})\ \boldsymbol{c}_{PR} = \boldsymbol{b}_{PR}$ for $\boldsymbol{c}_{PR}$. After a batch of edge insertions to the graph, the static algorithm to obtain the updated Katz scores first recomputes $\boldsymbol{x}_{t+1,Katz}$ using an iterative solver and obtains $\boldsymbol{c}_{t+1,Katz}$ as $A_{t+1}\boldsymbol{x}_{t+1,Katz}$ (for Katz Centrality) or recomputes $\boldsymbol{c}_{t+1,PR}$ (for personalized PageRank). For Katz Centrality, since calculating $\boldsymbol{c}_t$ given $\boldsymbol{x}_t$ at any time point t is one matrix-vector multiplication and can be done in $\mathcal{O}(m)$, this is not the bottleneck of static recomputation. Instead, the bottleneck is repeatedly updating $\boldsymbol{x}_{t,Katz}$ given more edges being inserted into the graph, and hence we focus our dynamic algorithm on limiting the number of iterations taken to obtain the updated vector $\boldsymbol{x}_{t,Katz}$, and, similarly, for PageRank, the focus of our dynamic algorithm on limiting the number of iterations taken to obtain the updated vector $\boldsymbol{c}_{t,PR}$. Therefore, for Katz, we solve for the correction $\Delta\boldsymbol{x}$ so that we can obtain the new solution at time $t+1$ as $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \Delta\boldsymbol{x}$. For PageRank, we solve for the correction $\Delta\boldsymbol{c}$ so that we can obtain the new solution at time $t+1$ as $\boldsymbol{c}_{t+1} = \boldsymbol{c}_t + \Delta\boldsymbol{c}$. The algorithm for updating Katz Centrality was previously derived in Chapter 4; we reproduce it here for clarity. The algorithm for updating PageRank is drawn from [17].

The first step in updating the centrality vector is to measure how close the old solution $\boldsymbol{x}_{t,Katz}$ or $\boldsymbol{c}_{t,PR}$ is to solving the system for the updated graph A_{t+1}. We

calculate the new residual for Katz Centrality as $\widetilde{\boldsymbol{r}}_{t+1,Katz}$

$$\begin{aligned}
\widetilde{\boldsymbol{r}}_{t+1,Katz} &= \boldsymbol{b}_{Katz} - M_{t+1,Katz}\boldsymbol{x}_{t,Katz} \\
&= \boldsymbol{b}_{Katz} - (I - \alpha A_{t+1})\boldsymbol{x}_{t,Katz} \\
&= \boldsymbol{b}_{Katz} - \boldsymbol{x}_{t,Katz} + \alpha A_{t+1}\boldsymbol{x}_{t,Katz} \\
&= \boldsymbol{b}_{Katz} - \boldsymbol{x}_{t,Katz} + \alpha A_t\boldsymbol{x}_{t,Katz} - \alpha A_t\boldsymbol{x}_{t,Katz} + \alpha A_{t+1}\boldsymbol{x}_{t,Katz} \\
&= \boldsymbol{r}_{t,Katz} + \alpha(A_{t+1} - A_t)\boldsymbol{x}_{t,Katz} \\
&= \boldsymbol{r}_{t,Katz} + \alpha\Delta A\boldsymbol{x}_{t,Katz}.
\end{aligned}$$

Similarly, for PageRank, we calculate the new residual as $\widetilde{\boldsymbol{r}}_{t+1,PR}$ (note since we use undirected graphs, $A^T = A$):

$$\begin{aligned}
\widetilde{\boldsymbol{r}}_{t+1,PR} &= \boldsymbol{b}_{PR} - M_{t+1,PR}\boldsymbol{c}_{t,PR} \\
&= (1-\alpha)\boldsymbol{v} - \boldsymbol{c}_{t,PR} + \alpha A_{t+1}D_{t+1}^{-1}\boldsymbol{c}_{t,PR} \\
&= (1-\alpha)\boldsymbol{v} - \boldsymbol{c}_{t,PR} + \alpha AD^{-1}\boldsymbol{c}_{t,PR} - \alpha AD^{-1}\boldsymbol{c}_{t,PR} + \alpha A_{t+1}D_{t+1}^{-1}\boldsymbol{c}_{t,PR} \\
&= \boldsymbol{r}_{t,PR} + \alpha(A_{t+1}D_{t+1}^{-1} - AD^{-1})\boldsymbol{c}_{t,PR}.
\end{aligned}$$

For both centrality measures, $\widetilde{\boldsymbol{r}}_{t+1}$ can be written in terms of the current residual at time t, edge updates ΔA, and the old solution. Next, we can use $\widetilde{\boldsymbol{r}}_{t+1}$ to set up a linear system for the correction $\Delta\boldsymbol{x}$ or $\Delta\boldsymbol{c}$. We apply iterative refinement [102] and for Katz Centrality, solve the linear system

$$(I - \alpha A_{t+1})\boldsymbol{\Delta x} = \widetilde{\boldsymbol{r}}_{t+1,Katz} = \boldsymbol{r}_{t,Katz} + \alpha\Delta A\boldsymbol{x}_{t,Katz}$$

for $\Delta\boldsymbol{x}$. For PageRank, we solve the linear system

$$(I - \alpha A_{t+1}D_{t+1}^{-1})\boldsymbol{\Delta c} = \widetilde{\boldsymbol{r}}_{t+1,PR} = \boldsymbol{r}_{t,PR} + \alpha(A_{t+1}D_{t+1}^{-1} - AD^{-1})\boldsymbol{c}_{t,PR}$$

for $\Delta \boldsymbol{c}$. Unlike iterative refinement's typical use of a directly factored matrix, we rely on Jacobi iteration for solving the above systems.

The final step of our algorithm is to update the residuals $\boldsymbol{r}_{t+1,Katz}$ and $\boldsymbol{r}_{t+1,PR}$ for the next time point. For Katz centrality, we can write the new residual $\boldsymbol{r}_{t+1,Katz}$ as

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

We can calculate $\boldsymbol{\Delta r}$, the difference in the two residuals at time t and $t+1$ as $\Delta\, \boldsymbol{r} = \alpha \Delta A \boldsymbol{x}_{t,Katz} - (I\, -\, \alpha\, A_{t+1})\boldsymbol{\Delta x}$. Likewise, updating the residual for PageRank is very similar. We can write the new residual $\boldsymbol{r}_{t+1,PR}$ as

$$\begin{aligned}
\boldsymbol{r}_{t+1,PR} &= (1-\alpha)\boldsymbol{v} - (I - \alpha A_{t+1} D_{t+1}^{-1})(\boldsymbol{c}_{t,PR} + \Delta \boldsymbol{c}) \\
&= (1-\alpha) - (I - \alpha A_{t+1} D_{t+1}^{-1})\boldsymbol{c}_{t,PR} - (I - \alpha A_{t+1} D_{t+1}^{-1})\boldsymbol{\Delta c} \\
&= \widetilde{\boldsymbol{r}}_{t+1,PR} - (I - \alpha A_{t+1} D_{t+1}^{-1})\Delta \boldsymbol{c} \\
&= \boldsymbol{r}_{t,PR} + \alpha(A_{t+1} D_{t+1}^{-1} - A_t D_t^{-1})\boldsymbol{c}_{t,PR} - (I - \alpha A_{t+1} D_{t+1}^{-1})\Delta \boldsymbol{c}.
\end{aligned}$$

Then, we can calculate $\boldsymbol{\Delta r}$, the difference in the two residuals at time t and $t+1$ as $\Delta \boldsymbol{r} = \alpha(A_{t+1} D_{t+1}^{-1} - A_t D_t^{-1})\boldsymbol{c}_{t,PR} - (I - \alpha A_{t+1} D_{t+1}^{-1})\Delta \boldsymbol{c}$. Updating the residual comes with the potential issue of accumulating error over long periods of time. However, these cases are rare, and, for our purposes, we obtain accurate results using our methods compared to a pure static recomputation. In Sections 5.3.2 and 5.3.3, we show that our algorithm for dynamic Katz Centrality maintains good quality of the updated scores for our community detection purposes and provides significant

speedup compared to a pure static recomputation in Section 5.3.3.

5.3.2 Synthetic Dynamic Graphs

In this section, we evaluate our dynamic algorithm on a synthetic network to show our ability to track merging and splitting of communities. We use a synthetic stochastic block model network with a recursive matrix (R-MAT) background with parameters $a = 0.55, b = 0.15, c = 0.15, d = 0.25$. Recall that an R-MAT generator [80] creates scale-free networks designed to emulate real-world networks. For an adjacency matrix, the matrix is subdivided into four quadrants, where each quadrant has a different probability of being selected where the probability of selecting each quadrant is given by a, b, c, d respectively. Once a quadrant is selected, this quadrant is recursively subdivided into four subquadrants and using the previous probabilities, we select one of the subquadrants. This process is repeated until we arrive at a single cell in the adjacency matrix. An edge is assigned between the two vertices making up that cell.

Figure 5.2 shows the community evolution in the stochastic block model part of the synthetic network that we are able to track with our dynamic algorithm. In Figure 5.2a, we start with three separate communities: C_1 (the top left community), C_2 (the middle community), and C_3 (the bottom right community). In Figure 5.2b, communities C_1 and C_2 merge together, and, in Figure 5.2c, C_1 splits off but communities C_2 and C_3 are merged together. Finally, in Figure 5.2d, communities C_2 and C_3 split and we obtain the original graph of three disjoint communities.

We pick five seeds at random from community C_2 to track both merging and splitting of communities and evaluate the recall of the community produced by our dynamic algorithm compared to the ground truth community at each of the four time points and results are averaged over the different seeds. We test our algorithm on communities of size 100 and 1000. The entire graph (including the R-MAT background) is 1,048,576 vertices and 10,485,760 edges (edge factor of 10).

To generate dynamic stochastic block models, we use parameters $p_{in} \in \{0.2, 0.5\}$ and $p_{out} = 0.01$, where p_{in} and p_{out} are the probabilities of an edge existing between a pair of vertices that are in the same and different communities, respectively. These parameters ease describing communities that change size compared to parameters used in Section 5.2.2. For example, when a community grows, the average vertex degree would have to change in order to reflect the same p_{in} and p_{out} parameters.

At each time point, we compare the community obtained from static recomputation versus the community obtained from our dynamic algorithm. We track the changing centrality vector and select top R vertices as the community, where R is the expected size of the community given the synthetic example in Figure 5.2. For a community of size R, let C_S be the community obtained from the statically computed centrality vector (i.e., the top R highly ranked vertices from $\boldsymbol{c}_S$). Similarly, let C_D be the community obtained from the dynamically computed vector $\boldsymbol{c}_D$. We calculate the recall of the vertices in the community produced by the dynamic algorithm as

$$recall = \frac{|C_S \cap C_D|}{R}.$$

Table 5.2 gives the recall values at each time point for the different graphs tested with the averages for each time point at the bottom. In a majority of the time steps, we obtain a recall above 0.80 and the communities with 1000 vertices have a higher recall than the communities with 100 vertices. Therefore, we are able to track evolving communities over time in dynamic synthetic graphs.

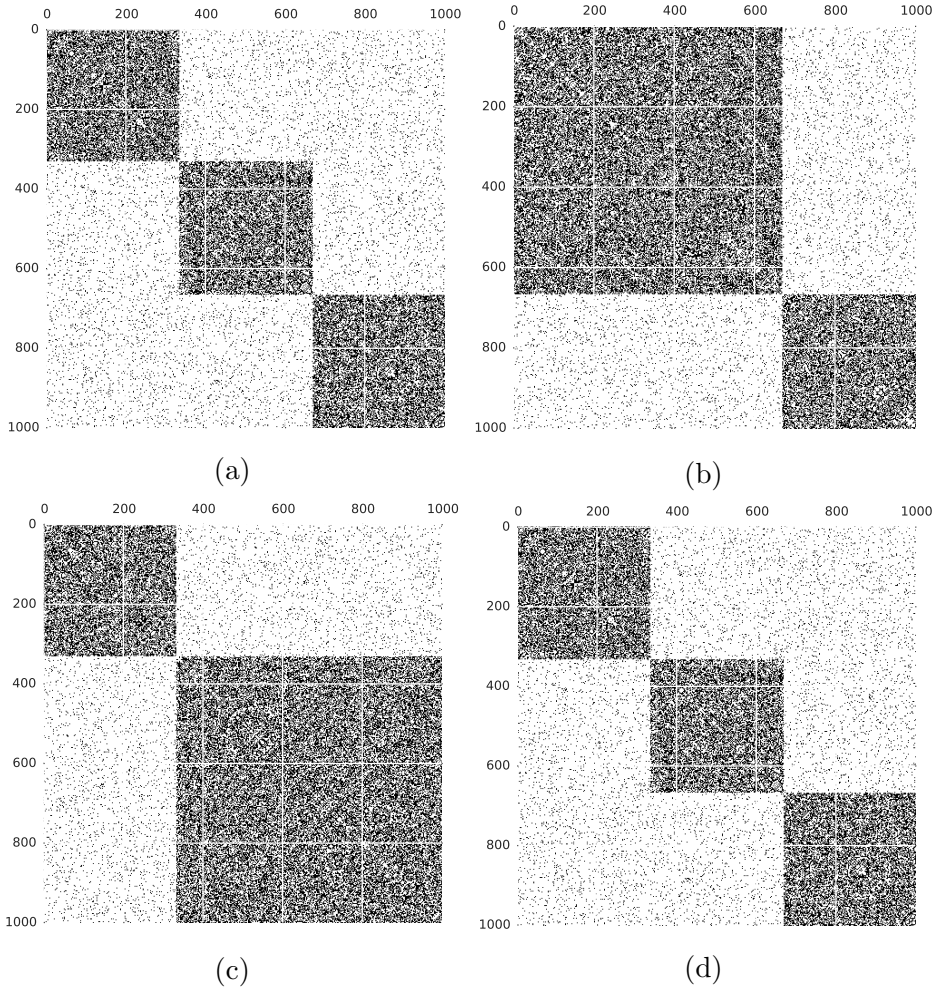


Figure 5.2: Synthetic dynamic graph showing merging and splitting of communities. (**a**) $t = 1$, (**b**) $t = 2$, (**c**) $t = 3$, (**d**) $t = 4$.

5.3.3 Real Graphs

We test our dynamic algorithm on five real graphs given in Table 5.3 from the KONECT database [76]. These graphs are chosen because they have timestamps associated with the edges in the graph. Since we have no ground truth of communities in real graphs, we use the results of the static algorithm as a pseudo-ground truth. Thus, every time we update the centrality scores using our dynamic algorithm, we recompute the centrality vector statically from scratch to have a baseline for comparison. We create an initial graph G_0 using the first half of edges, which provides a

Table 5.2: Average recalls at each point in time for synthetic merging and splitting of communities over time.

Parameters	**Batch Size**	$t =$	**Block Size = 100**				**Block Size = 1000**			
		$R =$	**1** **100**	**2** **200**	**3** **200**	**4** **100**	**1** **1000**	**2** **2000**	**3** **2000**	**4** **1000**
p_{in} =	10		0.86	0.94	0.93	0.85	0.93	0.97	0.98	9.98
0.2, p_{out}	100		0.76	0.89	0.89	0.75	0.97	0.99	0.99	0.97
= 0.01	1000		0.76	0.84	0.84	0.66	0.93	0.97	0.97	0.93
p_{in} =	10		0.92	0.96	0.97	0.92	0.96	0.98	0.99	0.99
0.5, p_{out}	100		0.79	0.89	0.90	0.78	0.95	0.98	0.98	0.95
= 0.01	1000		0.88	0.91	0.91	0.82	0.96	0.99	0.99	0.96
	Average		0.83	0.91	0.91	0.80	0.95	0.98	0.98	0.96

starting point for both the dynamic and static algorithms.

To simulate a stream of edges in a dynamic graph, we insert the remaining edges in timestamped order in batches of size b and apply both algorithms and use batches of size $b = 10$, 100, and 1000. We use communities of size $R \in \{100, 1000\}$.

Table 5.3: Real graphs used in experiments. Columns are graph name, number of vertices, and number of edges.

Graph	$\|\boldsymbol{V}\|$	$\|\boldsymbol{E}\|$
slashdot-threads	51,083	140,778
enron	87,221	1,148,072
digg	279,630	1,731,653
wiki-talk	541,355	2,424,962
youtube-u-growth	3,223,589	9,375,374

We evaluate our dynamic algorithm with respect to performance and quality. For performance, we calculate the speedup with respect to time and iterations. Denote the time taken by static recomputation and our dynamic algorithm as T_S and T_D, respectively. Similarly, denote the number of iterations taken by static recomputation and our dynamic algorithm as I_S and I_D, respectively. We then calculate speedups in time and iterations as:

$$speedup_{time} = \frac{T_S}{T_D}, speedup_{iter} = \frac{I_S}{I_D}.$$

Higher values of the speedups indicate that our dynamic algorithm provides more benefits compared to a pure static recomputation. We evaluate the quality of the results produced by our algorithms using three metrics: (1) recall, (2) ratio of conductance, and (3) ratio of normalized edge cut.

We denote the conductance (ϕ) of the community obtained from static recomputation as ϕ_S and the conductance of the community obtained from our dynamic

algorithm as ϕ_D, so we calculate the ratio of conductance scores as $\frac{\phi_S}{\phi_D}$. Since lower values of conductance indicate higher quality communities, a ratio of conductance scores greater than 1 indicates our dynamic algorithm produces higher quality communities than static recomputation. We denote the normalized edge cut for the community (f) obtained by the static recomputation as f_S and f_D for the community obtained from our dynamic algorithm. We calculate the ratio of cuts as $\frac{f_S}{f_D}$, where values of the ratio in scores less than 1 indicate that our dynamic algorithm produces higher quality communities than static recomputation.

We first show averages over time for all batch sizes for all graphs tested for all the performance and quality metrics in Table 5.4. Results shown are averaged over both community sizes tested. Unless otherwise specified, we use a single seed vertex and average over five different seeds for the personalized centrality metric. For a majority of the graphs, most notably the three larger graphs, the speedup in both time and number of iterations does not decrease with increasing batch size. This shows that our algorithm is able to maintain significant speedups even with large batch insertions of up to 1000 edges. We note that our dynamic algorithm also produces high quality communities compared to static recomputation. In terms of the recall, our dynamic algorithm always has recall values greater than 0.85, meaning we correctly identify a majority of the vertices in the local community compared to static recomputation, regardless of the batch size. Next, we examine the ratio of conductance scores of the community obtain via static recomputation compared to the community obtained from our dynamic algorithm. Ratios close to 1 indicate that the communities produced from our dynamic algorithm are similar to the ones produced from static recomputation w.r.t. their conductance scores, and values greater than 1 indicate our dynamic algorithm produces higher quality communities than static recomputation. In a majority of the graphs and batch sizes tested, we obtain ratios very close to 1, and in some cases ratios greater than 1. Since we treat static recom-

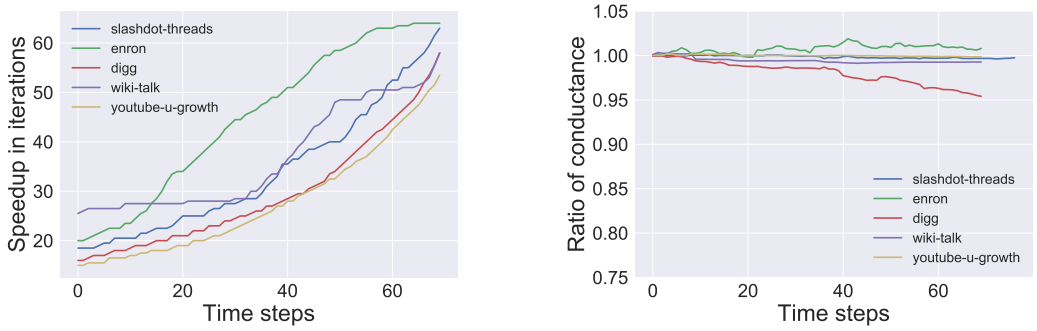
putation as ground truth, this means that the dynamic communities are of similar quality to the static communities, and, in some cases, higher quality than the statically computed ones. Finally, we compare values of the normalized edge cut for both communities. Recall that ratios of the normalized edge cut lower than 1 indicate that our dynamic algorithm produces higher quality communities w.r.t. the normalized edge cut. In a majority of cases, we obtain communities close in quality to that of static recomputation, similar to the results we see from comparing the conductances of the communities obtained from static recomputation and our dynamic algorithm. In summary, the most prominent trends from this table are twofold: (1) we see significant speedups w.r.t. both time and iteration counts by using our dynamic algorithm compared to static recomputation to compute local communities using personalized centrality metrics, and (2) the communities produced by our dynamic algorithm are of similar quality to that of static recomputation, and in some cases, of higher quality.

Next, we examine the performance and quality of our algorithm over time. Figure 5.3 plots the speedup in iterations over time (Figure 5.3a) and the ratio of conductance scores over time (Figure 5.3b). Since our dynamic algorithm only targets places in the centrality vector that are directly affected by edge updates to the graph, the performance of our dynamic algorithm is unaffected by the size of the underlying graph. This is unlike static recomputation, which is directly affected by the size of the underlying graph. Therefore, the speedup in iterations increases over time. Finally, we observe that the quality of our dynamic algorithm (in terms of conductance) matches the quality of the static algorithm with little to no decrease over time. There is only one graph for which the quality slightly decreases over time (DIGG); however, even for this graph, the ratio of conductances scores is still consistently above 0.95. In contrast, for the ENRON graph, the conductance of the dynamic community is better than the conductance of the static community (ratios greater than 1). These results show that our dynamic algorithm helps more in terms of performance over time,

Table 5.4: Average summary statistics over time on real graphs for all batch sizes. Columns are graph name, batch size, speedup in time, speedup in iterations, recall, ratio of conductance scores, and ratio of normalized edge cut scores.

Graph	Batch Size	Performance		Quality		
		T_S/T_D	I_S/I_D	Recall	ϕ_S/ϕ_D	f_S/f_D
slashdot-threads	10	52.94×	34.02×	0.93	0.99	1.03
	100	26.88×	21.46×	0.96	1.00	1.01
	1000	39.65×	31.09×	0.96	1.00	1.00
enron	10	75.42×	45.04×	0.97	1.00	1.00
	100	63.61×	41.28×	0.98	1.01	0.98
	1000	46.20×	29.57×	0.96	1.01	0.98
digg	10	54.29×	29.41×	0.86	0.97	1.18
	100	47.64×	25.69×	0.90	0.98	1.07
	1000	50.64×	26.87×	0.97	0.99	1.02
wiki-talk	10	56.02×	36.68×	0.95	1.00	1.02
	100	48.87×	31.46×	0.91	0.99	1.19
	1000	56.22×	36.95×	0.96	1.00	1.02
youtube- u-growth	10	56.47×	27.66×	0.96	1.00	0.94
	100	50.00×	26.58×	0.96	1.00	1.00
	1000	40.17×	20.44×	0.91	1.00	0.92

without sacrificing the quality of the communities produced.



(a) (b)

Figure 5.3: Performance and quality behavior of dynamic algorithm compared to static recomputation over time. (**a**) Speedup in iterations over time for $b = 10$, (**b**) Ratio of conductance scores over time for $b = 100$.

Different Seeding Methods

Finally, we examine different methods of choosing seed vertices. The purpose of this section is to test our dynamic algorithm with multiple seeds used in the right-hand side vector. All the previous results have been w.r.t. a single seed vertex (or averages of results for single seeds) chosen randomly from a pool of the top 10% highest degree vertices. We now use the following three methods to choose multiple seeds: (1) RW-1, (2) RW-2, and (3) RW-3, similar to [103]. Using just one seed vertex i, the right-hand side $\boldsymbol{b} = \boldsymbol{e}_i$. In the case of using multiple seeds $S = \{v_1, v_2, \cdots, v_{|S|}\}$, the right hand side is $\boldsymbol{b} = \boldsymbol{e}_{v_1} + \boldsymbol{e}_{v_1} + \cdots + \boldsymbol{e}_{v_{|S|}}$. The method RW-$k$ chooses $|S|$ seeds as follows: we first choose a vertex v at random from the existing vertices in the initial graph. We perform a random walk of length k from v and take the terminal vertex as a seed. We repeat this procedure to generate $|S|$ unique seeds. Table 5.5 gives the results for different seeds methods for all the evaluation metrics. With respect to the speedup in both time and iterations, there is no significant difference in using a larger number of seed vertices or a different seeding methods. This intuitively makes sense since varying the number of seeds merely changes the right-hand side vector $\boldsymbol{b}$ of the linear system, which has no effect on how many iterations the iterative solver takes to converge to a solution. With regards to the recall, we see a slight increase in recall values for a larger number of seeds. This can be attributed to the fact that a larger number of seeds indicates that the right-hand side vector has a larger number of nonzero values, meaning that the centrality values produced ($\boldsymbol{c}$) are with respect to all the seeds instead a single one.

Furthermore, the method RW-1 produces higher values of recall than RW-2, which produces higher values of recall than RW-3 across multiple number of seed vertices. We offer a possible explanation for this behavior. The seeds produced by RW-1 are all neighbors of a single vertex and are more likely to belong to the same community. Since the seeds produced by RW-3 are three steps away from the initial randomly

chosen vertex, it is less likely that these seeds belong to the same community, so the highly ranked vertices in the personalized centrality metric with respect to these seeds may not be as tightly knit of a community. However, the ratios of the conductance and normalized edge cut see no significant differences in varying the number of seeds or different seeding methods. All three seeding methods produce similar quality communities from our dynamic algorithm compared to static recomputation. In summary, we see that the performance doesn't change with respect to the number of seeds used, but the quality in terms of the recall shows a slight increase with more seeds used.

5.4 Conclusions

The problems of community detection and centrality have been well-studied in recent years. In this work, we have bridged these two fields by presenting a new method of identifying local communities using personalized centrality metrics.

We extended previous work in [6, 17] by using dynamic algorithms for calculating centrality scores in order to find local communities in evolving networks.

Our method uses the top R highly ranked vertices from a personalized centrality metric as the local community with respect to seed vertices of interest. Experiments on synthetic networks show that our method is able to identify blocks in artificially generated stochastic block models. We have shown that we obtain a high recall of the vertices using our method compared to the ground truth and that our method is faster than conventional local community detection methods such as greedy seed set expansion. Next, we extended our method to detect evolving communities in dynamic graphs. Using a synthetic example of a stochastic block model graph overlaid on an R-MAT background, we showed that our method successfully detects merging and splitting of communities over time. We applied our methods to real graphs and showed that our algorithm yields similar quality communities to static recomputation and is faster in both time and the number of iterations taken.

Metric	Method	Number of Seeds									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
T_S/T_D	RW-1	46.9×	54.4×	49.3×	41.7×	39.4×	30.3×	32.4×	47.3×	41.5×	29.3×
	RW-2	33.7×	66.1×	42.8×	51.5×	57.0×	52.1×	50.6×	46.1×	53.2×	39.0×
	RW-3	44.5×	53.4×	54.0×	44.3×	53.6×	44.5×	53.0×	63.2×	68.5×	47.8×
I_S/I_D	RW-1	29.4×	30.9×	29.8×	24.6×	24.5×	24.4×	21.0×	29.2×	25.3×	22.3×
	RW-2	20.4×	37.3×	24.4×	30.9×	31.8×	29.2×	29.0×	28.4×	30.1×	24.1×
	RW-3	26.0×	29.8×	31.9×	27.9×	33.4×	27.4×	30.9×	38.2×	37.0×	29.9×
Recall	RW-1	0.99	0.98	1.00	0.98	1.00	1.00	0.99	0.98	1.00	1.00
	RW-2	0.96	0.98	0.95	0.99	0.96	0.99	1.00	0.99	0.99	0.99
	RW-3	0.93	0.97	0.95	0.99	0.98	0.99	0.99	0.98	0.99	0.99
ϕ_S/ϕ_D	RW-1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	RW-2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	RW-3	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
f_S/f_D	RW-1	0.99	0.98	1.00	1.00	1.00	1.00	0.94	1.01	0.98	1.00
	RW-2	1.00	0.97	0.99	1.01	1.03	1.00	1.00	1.00	1.00	1.00
	RW-3	1.03	0.99	1.01	1.00	1.00	1.00	1.00	1.00	1.00	1.01

Table 5.5: Results for different seeding methods. Columns are metric tested, seeding method, speedup in time, speedup in iterations, recall, ratio of conductance scores, and ratio of normalized edge cut scores. Results shown are averaged over all graphs.

The main drawback of our method is that it requires previous knowledge of the community size. Future work can investigate methods to identify the local community using personalized centrality without previous knowledge of community size; however this is out of the scope of this dissertation. For example, we can use a sweep cut method similar to [98]. After the personalized centrality metric is calculated using our dynamic algorithm, we can sweep over all cuts induced by the ordering of the personalized centrality vector and choose the best cut determined by conductance scores of the induced cuts. This method would therefore identify the best local community given the centrality scores without any size requirement. Since local communities with respect to a seed vertex can be computed independently of other local communities with respect to different seeds, the computation of separate communities can be easily parallelized. By computing multiple local communities across an entire graph, we can partition the global graph into different communities. This can also be addressed in future work.

CHAPTER 6
CONCLUSIONS

This dissertation presented numerical and streaming techniques for the analysis of centrality measures in graphs. We provided fast, efficient, and theoretically correct algorithms to compute centrality metrics in both static and dynamic graphs and concluded with an application of centrality to the equally well-studied graph query of community detection.

Chapter 3 bridges the gap between numerical analysis and data analysis. We developed theory and presented techniques to better understand how error in a numerical problem can translate to error in a corresponding data analysis problem. Specifically we studied how error present from an approximation to the solution to a linear system from an iterative method can guarantee accurate ranking of vertices in a graph. This led to the development of a new stopping criterion that can be used in conjunction with any iterative method. We show how to obtain the exact highly ranked vertices upon termination at this new stopping criterion, regardless of whatever numerical accuracy we obtain at this point. When identifying highly ranked vertices with Katz Centrality and PageRank, results show that our method is able to not only reduce the number of iterations taken by an iterative solver compared to commonly used techniques but also provide previously missing theoretical guarantees of correctness of the vertices returned.

While Chapter 3 focused exclusively on static graphs, in Chapter 4 we moved into the realm of dynamic graphs and presented four dynamic algorithms for computations of different centrality metrics. First we presented two algorithms for calculating Katz Centrality scores in dynamic graphs. The first algorithm exploits properties of iterative solvers and updates the linear algebraic formulation of Katz Centrality in order to

obtain updated scores of vertices in evolving networks. The second algorithm focuses on personalized Katz scores, or scores with respect to specific seed vertices of interest. We moved away from linear algebraic computations and presented an agglomerative algorithm by starting with the seed vertex and iteratively calculating scores for successively larger neighborhoods of vertices. We shifted focus from walk-based Katz Centrality to develop a new dynamic algorithm for a centrality metric based on non-backtracking walks, where these walks do not allow backtracking sequences such as $0 \to 1 \to 0$. The last dynamic algorithm for centrality presented is for exponential-based centrality values. By exploiting properties of matrix exponentials, we were able to derive a new algorithm to take advantage of previous timesteps' computations to aid in our computation of the matrix exponential in the current timestep. Each of the algorithms presented was compared to its static counterpart: the naive algorithm of continuously recalculating centrality scores from scratch everytime the underlying graph is changed. For each dynamic algorithm, we obtained significant speedups in time (and when applicable, in iterations from an iterative solver). Furthermore, our algorithms all preserve the quality of scores (meaning they return similar numerical scores) when compared to a pure static recomputation.

Finally, Chapter 5 applied techniques derived for the dynamic computation of centrality to the task of identifying local communities in dynamic graphs. We studied how we can use the resultant ranking of vertices from a personalized centrality vector to track the respective local community in a dynamic graph. Our centrality based community detection algorithm is able to find similar quality communities to that of a commonly used agglomerative community detection algorithm. Results on both synthetic and real-world networks show that our dynamic algorithm is several orders of magnitude faster than a pure static recomputation.

In this dissertation, we have answered questions about the computation of centrality from both a numerical and a dynamic standpoint. These contributions open

up a variety of avenues for future research, such as parallelization of the algorithms discussed. The techniques provided in this thesis give new insight into different kinds of analyses that can be performed when analyzing large datasets.

REFERENCES

[1] M. Benzi, E. Estrada, and C. Klymko, "Ranking hubs and authorities using matrix functions," *Linear Algebra and its Applications*, vol. 438, no. 5, pp. 2447–2474, 2013.

[2] E. Nathan, G. Sanders, J. Fairbanks, V. E. Henson, and D. A. Bader, "Graph ranking guarantees for numerical approximations to katz centrality," *Procedia Computer Science*, vol. 108, pp. 68–78, 2017.

[3] E. Nathan, G. Sanders, V. E. Henson, and D. Bader, "Numerically approximating centrality for graph ranking guarantees," *Journal of computational science*, 2018.

[4] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.

[5] D. F. Gleich, "Pagerank beyond the web," *SIAM Review*, vol. 57, no. 3, pp. 321–363, 2015.

[6] E. Nathan and D. 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*, ACM, 2017.

[7] E. Nathan and D. A. Bader, "Incrementally updating personalized katz centrality in dynamic graphs," *Social Network Analysis and Mining*, 2018.

[8] ——, "Approximating personalized katz centrality in dynamic graphs," in *International Conference on Parallel Processing and Applied Mathematics*, Springer, 2017.

[9] E. Nathan, J. Fairbanks, and D. Bader, "Ranking in dynamic graphs using exponential centrality," in *International Workshop on Complex Networks and their Applications*, Springer, 2017, pp. 378–389.

[10] E. Nathan, A. Zakrzewska, J. Riedy, and D. A. Bader, "Local community detection in dynamic graphs using personalized centrality," *Algorithms*, vol. 6, no. 1, p. 65, 2017.

[11] M. Newman, *Networks: an introduction*. Oxford university press, 2010.

[12] M. Hofmeister, "Spectral radius and degree sequence," *Mathematische Nachrichten*, vol. 139, no. 1, pp. 37–44, 1988.

[13] D. A. Spielman, "Algorithms, graph theory, and linear equations in laplacian matrices," in *Proceedings of the International Congress of Mathematicians 2010 (ICM 2010) (In 4 Volumes) Vol. I: Plenary Lectures and Ceremonies Vols. II–IV: Invited Lectures*, World Scientific, 2010, pp. 2698–2722.

[14] R. Albert, H. Jeong, and A.-L. Barabási, "The diameter of the world wide web," *arXiv preprint cond-mat/9907038*, 1999.

[15] O. Livne and A. Brandt, "Lean algebraic multigrid (lamg): fast graph laplacian linear solver," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, B499–B522, 2012.

[16] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: bringing order to the web.," 1999.

[17] J. Riedy, "Updating PageRank for streaming graphs," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, IEEE, 2016, pp. 877–884.

[18] P. Bonacich, "Some unique properties of eigenvector centrality," *Social networks*, vol. 29, no. 4, pp. 555–564, 2007.

[19] O. Perron, "Zur theorie der matrices," *Mathematische Annalen*, vol. 64, no. 2, pp. 248–263, 1907.

[20] E. Estrada and J. A. Rodriguez-Velazquez, "Subgraph centrality in complex networks," *Physical Review E*, vol. 71, no. 5, p. 056 103, 2005.

[21] M. Benzi and C. Klymko, "Total communicability as a centrality measure," *Journal of Complex Networks*, vol. 1, no. 2, pp. 124–149, 2013.

[22] D. Werner, *Funktionalanalysis*. Springer, 2006.

[23] E. Kokiopoulou, J. Chen, and Y. Saad, "Trace optimization and eigenproblems in dimension reduction methods," *Numerical Linear Algebra with Applications*, vol. 18, no. 3, pp. 565–602, 2011.

[24] J. P. Fairbanks, A. Zakrzewska, and D. A. Bader, "New stopping criteria for spectral partitioning," in *Advances in Social Networks Analysis and Mining (ASONAM), 2016 IEEE/ACM International Conference on*, IEEE, 2016, pp. 25–32.

[25] U. Brandes and C. Pich, "Centrality estimation in large networks," *International Journal of Bifurcation and Chaos*, vol. 17, no. 07, pp. 2303–2318, 2007.

[26] S. White and P. Smyth, "Algorithms for estimating relative importance in networks," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2003, pp. 266–275.

[27] K. C. Foster, S. Q. Muth, J. J. Potterat, and R. B. Rothenberg, "A faster katz status score algorithm," *Computational & Mathematical Organization Theory*, vol. 7, no. 4, pp. 275–285, 2001.

[28] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *Journal of the American society for information science and technology*, vol. 58, no. 7, pp. 1019–1031, 2007.

[29] F. Bonchi, P. Esfandiar, D. F. Gleich, C. Greif, and L. V. Lakshmanan, "Fast matrix computations for pairwise and columnwise commute times and katz scores," *Internet Mathematics*, vol. 8, no. 1-2, pp. 73–112, 2012.

[30] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.

[31] R. W. Floyd, "Algorithm 97: shortest path," *Commun. ACM*, vol. 5, pp. 345–345, 6 1962.

[32] S. Warshall, "A theorem on boolean matrices," *J. ACM*, vol. 9, pp. 11–12, 1 1962.

[33] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 1–13, 1977.

[34] U. Brandes, "A faster algorithm for betweenness centrality*," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[35] A. Bavelas, "Communication patterns in task-oriented groups.," *Journal of the acoustical society of America*, 1950.

[36] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.

[37] D. Eppstein and J. Wang, "Fast approximation of centrality.," *J. Graph Algorithms Appl.*, vol. 8, pp. 39–45, 2004.

[38] K. Okamoto, W. Chen, and X.-Y. Li, "Ranking of closeness centrality for large-scale social networks," in *Frontiers in Algorithmics*, Springer, 2008, pp. 186–195.

[39] W. Wei and K. Carley, "Real time closeness and betweenness centrality calculations on streaming network data," 2014.

[40] A. E. Sariyüce, K. Kaya, E. Saule, and Ü. V. Çatalyiirek, "Incremental algorithms for closeness centrality," in *Big Data, 2013 IEEE International Conference on*, IEEE, 2013, pp. 487–492.

[41] O. Green, R. McColl, and D. Bader, "A fast algorithm for incremental betweenness centrality," in *Proceeding of SE/IEEE international conference on social computing (SocialCom)*, 2012, pp. 3–5.

[42] Y.-Y. Chen, Q. Gan, and T. Suel, "Local methods for estimating PageRank values," in *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, ACM, 2004, pp. 381–389.

[43] S. Chien, C. Dwork, R. Kumar, and D Sivakumar, "Towards exploiting link evolution," 2001.

[44] A. D. Sarma, S. Gollapudi, and R. Panigrahy, "Estimating PageRank on graph streams," *Journal of the ACM (JACM)*, vol. 58, no. 3, p. 13, 2011.

[45] Z. Gyöngyi, H. Garcia-Molina, and J. Pedersen, "Combating web spam with trustrank," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, VLDB Endowment, 2004, pp. 576–587.

[46] A. N. Langville and C. D. Meyer, "Updating PageRank using the group inverse and stochastic complementation," *Informe técnico crsc02-tr32*, 2002.

[47] A. N. Langville and C. D. Meyer, "Updating the stationary vector of an irreducible Markov chain with an eye on Googles PageRank," in *In SIMAX*, Citeseer, 2004.

[48] A. N. Langville and C. D. Meyer, "Updating PageRank with iterative aggregation," in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, ACM, 2004, pp. 392–393.

[49] B. Bahmani, A. Chowdhury, and A. Goel, "Fast incremental and personalized PageRank," *Proceedings of the VLDB Endowment*, vol. 4, no. 3, pp. 173–184, 2010.

[50] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical review E*, vol. 70, no. 6, p. 066 111, 2004.

[51] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, P10008, 2008.

[52] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM journal on matrix analysis and applications*, vol. 11, no. 3, pp. 430–452, 1990.

[53] F. R. Chung, *Spectral graph theory*. American Mathematical Soc., 1997, vol. 92.

[54] I. Derényi, G. Palla, and T. Vicsek, "Clique percolation in random networks," *Physical review letters*, vol. 94, no. 16, p. 160 202, 2005.

[55] J. Xie, B. K. Szymanski, and X. Liu, "Slpa: uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process," in *Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on*, IEEE, 2011, pp. 344–349.

[56] J. Xie and B. K. Szymanski, "Towards linear time overlapping community detection in social networks," in *Advances in Knowledge Discovery and Data Mining*, Springer, 2012, pp. 25–36.

[57] T. Evans and R Lambiotte, "Line graphs of weighted networks for overlapping communities," *The European Physical Journal B*, vol. 77, no. 2, pp. 265–272, 2010.

[58] A. Lancichinetti, F. Radicchi, J. J. Ramasco, and S. Fortunato, "Finding statistically significant communities in networks," *PloS one*, vol. 6, no. 4, e18961, 2011.

[59] A. Lancichinetti, S. Fortunato, and J. Kertész, "Detecting the overlapping and hierarchical community structure in complex networks," *New Journal of Physics*, vol. 11, no. 3, p. 033 015, 2009.

[60] F. Havemann, M. Heinz, A. Struck, and J. Gläser, "Identification of overlapping communities and their hierarchy by locally calculating community-changing resolution levels," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2011, no. 01, P01023, 2011.

[61] C. Lee, F. Reid, A. McDaid, and N. Hurley, "Detecting highly overlapping community structure by greedy clique expansion," in *4th SNA-KDD Workshop*, 2010, 3342.

[62] C. L. Staudt and H. Meyerhenke, "Engineering parallel algorithms for community detection in massive networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 171–184, 2016.

[63] C. Tantipathananandh, T. Berger-Wolf, and D. Kempe, "A framework for community identification in dynamic social networks," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2007, pp. 717–726.

[64] P. J. Mucha, T. Richardson, K. Macon, M. A. Porter, and J.-P. Onnela, "Community structure in time-dependent, multiscale, and multiplex networks," *science*, vol. 328, no. 5980, pp. 876–878, 2010.

[65] M. B. Jdidia, C. Robardet, and E. Fleury, "Communities detection and analysis of their dynamics in collaborative networks.," in *ICDIM*, 2007, pp. 744–749.

[66] D. Chakrabarti, R. Kumar, and A. Tomkins, "Evolutionary clustering," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2006, pp. 554–560.

[67] Y.-R. Lin, Y. Chi, S. Zhu, H. Sundaram, and B. L. Tseng, "Analyzing communities and their evolutions in dynamic social networks," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 3, no. 2, p. 8, 2009.

[68] T. Aynaud and J.-L. Guillaume, "Static community detection algorithms for evolving networks," in *WiOpt'10: Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, 2010, pp. 508–514.

[69] J. Shang, L. Liu, F. Xie, Z. Chen, J. Miao, X. Fang, and C. Wu, "A real-time detecting algorithm for tracking community structure of dynamic networks," *arXiv preprint arXiv:1407.2683*, 2014.

[70] T. N. Dinh, Y. Xuan, and M. T. Thai, "Towards social-aware routing in dynamic communication networks," in *Performance Computing and Communications Conference (IPCCC), 2009 IEEE 28th International*, IEEE, 2009, pp. 161–168.

[71] J. Riedy and D. A. Bader, "Multithreaded community monitoring for massive streaming graph data," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, IEEE, 2013, pp. 1646–1655.

[72] R. Görke, P. Maillard, A. Schumm, C. Staudt, and D. Wagner, "Dynamic graph clustering combining modularity and smoothness," *ACM Journal of Experimental Algorithmics*, vol. 18, 2013.

[73] R. Varga, *Gershgorin and his circles in springer series in computational mathematics, 36*, 2004.

[74] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[75] M. E. Hochstenbach, "Probabilistic upper bounds for the matrix two-norm," *Journal of Scientific Computing*, vol. 57, no. 3, pp. 464–476, 2013.

[76] J. Kunegis, "KONECT: the Koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*, ACM, 2013, pp. 1343–1350.

[77] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[78] J. H. Wilkinson, *Rounding errors in algebraic processes*. Courier Corporation, 1994.

[79] P. Erdös and A. Rényi, "On random graphs, i," *Publicationes Mathematicae (Debrecen)*, vol. 6, pp. 290–297, 1959.

[80] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: a recursive model for graph mining.," in *SDM*, SIAM, vol. 4, 2004, pp. 442–446.

[81] M. Benzi and C. Klymko, "A matrix analysis of different centrality measures," *arXiv preprint arXiv:1312.6722*, 2014.

[82] R. Albert, H. Jeong, and A.-L. Barabási, "Internet: diameter of the world-wide web," *Nature*, vol. 401, no. 6749, pp. 130–131, 1999.

[83] K. Hawick and H. James, "Node importance ranking and scaling properties of some complex road networks," 2007.

[84] P. Grindrod, D. J. Higham, and V. Noferini, "The deformed graph laplacian and its applications to network centrality analysis," *SIAM Journal on Matrix Analysis and Applications*, 2017.

[85] E. Estrada and D. J. Higham, "Network properties revealed through matrix functions," *SIAM review*, vol. 52, no. 4, pp. 696–714, 2010.

[86] N. J. Higham, *Functions of matrices: theory and computation*. SIAM, 2008.

[87] H. F. Trotter, "On the product of semi-groups of operators," *Proceedings of the American Mathematical Society*, vol. 10, no. 4, pp. 545–551, 1959.

[88] C. Moler and C. Van Loan, "Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later," *SIAM review*, vol. 45, no. 1, pp. 3–49, 2003.

[89] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[90] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-worldnetworks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[91] F. Bauer and J. T. Lizier, "Identifying influential spreaders and efficiently estimating infection numbers in epidemic models: a walk counting approach," *EPL (Europhysics Letters)*, vol. 99, no. 6, p. 68 007, 2012.

[92] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

[93] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026 113, 2004.

[94] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Graph partitioning and graph clustering, 10th dimacs implementation challenge workshop," *Contemporary Mathematics*, vol. 588, 2013.

[95] A. Clauset, "Finding local community structure in networks," *Physical review E*, vol. 72, no. 2, p. 026 132, 2005.

[96] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[97] J. P. Bagrow and E. M. Bollt, "Local method for detecting communities," *Physical Review E*, vol. 72, no. 4, p. 046 108, 2005.

[98] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using PageRank vectors," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, IEEE, 2006, pp. 475–486.

[99] F. Arrigo, P. Grindrod, D. J. Higham, and V. Noferini, "Nonbacktracking walk centrality for directed networks," University of Manchester, Tech. Rep. MIMS preprint 2017.9, Mar. 2017.

[100] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Statistical properties of community structure in large social and information networks," in *Proceedings of the 17th international conference on World Wide Web*, ACM, 2008, pp. 695–704.

[101] I. M. Kloumann, J. Ugander, and J. M. Kleinberg, "Block models and personalized pagerank," *CoRR*, vol. abs/1607.03483, 2016.

[102] C. B. Moler, "Iterative refinement in floating point," *J. ACM*, vol. 14, no. 2, pp. 316–321, 1967.

[103] J. Riedy, D. A. Bader, K. Jiang, P. Pande, and R. Sharma, "Detecting communities from given seeds in social networks," Georgia Institute of Technology, Tech. Rep., 2011.