Approximate Graph Propagation

Hanzhi Wang
hanzhi_wang@ruc.edu.cn
Renmin University of China
Beijing, China

Mingguo He
mingguo@ruc.edu.cn
Renmin University of China
Beijing, China

Zhewei Wei*
zhewei@ruc.edu.cn
Renmin University of China
Beijing, China

Sibo Wang
swang@se.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong, China

Ye Yuan
yuan-ye@bit.edu.cn
Beijing Institute of Technology
Beijing, China

Xiaoyong Du
Ji-Rong Wen
duyong,jrwen@ruc.edu.cn
Renmin University of China
Beijing, China

ABSTRACT

Efficient computation of node proximity queries such as transition probabilities, Personalized PageRank, and Katz are of fundamental importance in various graph mining and learning tasks. In particular, several recent works leverage fast node proximity computation to improve the scalability of Graph Neural Networks (GNN). However, prior studies on proximity computation and GNN feature propagation are on a case-by-case basis, with each paper focusing on a particular proximity measure.

In this paper, we propose Approximate Graph Propagation (AGP), a unified randomized algorithm that computes various proximity queries and GNN feature propagations, including transition probabilities, Personalized PageRank, heat kernel PageRank, Katz, SGC, GDC, and APPNP. Our algorithm provides a theoretical bounded error guarantee and runs in almost optimal time complexity. We conduct an extensive experimental study to demonstrate AGP's effectiveness in two concrete applications: local clustering with heat kernel PageRank and node classification with GNNs. Most notably, we present an empirical study on a billion-edge graph Papers100M, the largest publicly available GNN dataset so far. The results show that AGP can significantly improve various existing GNN models' scalability without sacrificing prediction accuracy.

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**; • **Information systems → Data mining**.

KEYWORDS

node proximity queries, local clustering, Graph Neural Networks

*Zhewei Wei is the corresponding author. Work partially done at Gaoling School of Artificial Intelligence, Beijing Key Laboratory of Big Data Management and Analysis Methods, MOE Key Lab DEKE, Renmin University of China, and Pazhou Lab, Guangzhou, 510330, China.

ACM Reference Format:
Hanzhi Wang, Mingguo He, Zhewei Wei, Sibo Wang, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. 2021. Approximate Graph Propagation. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3447548.3467243

1 INTRODUCTION

Recently, significant research effort has been devoted to compute *node proximty queries* such as Personalized PageRank [22, 32, 37, 38], heat kernel PageRank [12, 40] and the Katz score [23]. Given a node s in an undirected graph $G = (V, E)$ with $|V| = n$ nodes and $|E| = m$ edges, a node proximity query returns an n-dimensional vector $\boldsymbol{\pi}$ such that $\boldsymbol{\pi}(v)$ represents the importance of node v with respect to s. For example, a widely used proximity measure is the L-th transition probability vector. It captures the L-hop neighbors' information by computing the probability that a L-step random walk from a given source node s reaches each node in the graph. The vector form is given by $\boldsymbol{\pi} = \left(\mathbf{AD}^{-1}\right)^L \cdot \boldsymbol{e}_s$, where $\mathbf{A}$ is the adjacency matrix, $\mathbf{D}$ is the diagonal degree matrix with $\mathbf{D}(i,i) = \sum_{j\in V} \mathbf{A}(i,j)$, and $\boldsymbol{e}_s$ is the one-hot vector with $\boldsymbol{e}_s(s) = 1$ and $\boldsymbol{e}_s(v) = 0, v \neq s$. Node proximity queries find numerous applications in the area of graph mining, such as link prediction in social networks [6], personalized graph search techniques [21], fraud detection [4], and collaborative filtering in recommender networks [17].

In particular, a recent trend in Graph Neural Network (GNN) researches [25, 26, 39] is to employ node proximity queries to build scalable GNN models. A typical example is SGC [39], which simplifies the original Graph Convolutional Network (GCN) [24] with a linear propagation process. More precisely, given a self-looped graph and an $n \times d$ feature matrix $\mathbf{X}$, SGC takes the multiplication of the L-th normalized transition probability matrix $\mathbf{P} = \left(\mathbf{D}^{-\frac{1}{2}}\mathbf{AD}^{-\frac{1}{2}}\right)^L$ and the feature matrix $\mathbf{X}$ to form the representation matrix

$$\mathbf{Z} = \mathbf{P} \cdot \mathbf{X} = \left(\mathbf{D}^{-\frac{1}{2}} \cdot \mathbf{A} \cdot \mathbf{D}^{-\frac{1}{2}}\right)^L \cdot \boldsymbol{X}. \tag{1}$$

If we treat each column of the feature matrix $\mathbf{X}$ as a graph signal vector $\boldsymbol{x}$, then the representation matrix $\mathbf{Z}$ can be derived by the augment of d vectors $\boldsymbol{\pi} = \left(\mathbf{D}^{-\frac{1}{2}}\mathbf{AD}^{-\frac{1}{2}}\right)^L \cdot \boldsymbol{x}$. SGC feeds $\mathbf{Z}$ into a logistic regression or a standard neural network for downstream machine learning tasks such as node classification and link prediction.

The feature propagation matrix **P** can be easily generalized to other node proximity models, such as PPR used in APPNP [25], PPRGo [7] and GBP [10], and HKPR used in [26]. Compared to the original GCN [24] which uses a full-batch training process and stores the representation of each node in the GPU memory, these proximity-based GNNs decouple prediction and propagation and thus allows mini-batch training to improve the scalability of the models. Note that even though the ideas to employ PPR and HKPR models in the feature propagation process are borrowed from APPNP, PPRGo, GBP and GDC, the original papers of APPNP, PPRGo and GDC use extra complex structures to propagate node features. With a slight abuse of notation, we use APPNP and GDC to denote the linear propagation process by substituting **P** in equation (1) to PPR and HKPR models, respectively.

Graph Propagation. To model various proximity measures and GNN propagation formulas, we consider the following unified *graph propagation equation*:

$$\boldsymbol{\pi} = \sum_{i=0}^{\infty} w_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \boldsymbol{x}, \tag{2}$$

where **A** denotes the adjacency matrix, **D** denotes the diagonal degree matrix, a and b are the Laplacian parameters that take values in $[0,1]$, the sequence of w_i for $i = 0, 1, 2, \ldots$ is the weight sequence and $\boldsymbol{x}$ is an n dimensional vector. Following the convention of Graph Convolution Networks [24], we refer to $\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}$ as the *propagation matrix*, and $\boldsymbol{x}$ as the *graph signal vector*.

A key feature of the graph propagation equation (2) is that we can manipulate parameters a, b, w_i and $\boldsymbol{x}$ to obtain different proximity measures. For example, if we set $a = 0, b = 1, w_L = 1, w_i = 0$ for $i = 0, \ldots, L-1$, and $\boldsymbol{x} = \boldsymbol{e}_s$, then $\boldsymbol{\pi}$ becomes the L-th transition probability vector from node s. Table 1 summarizes the proximity measures and GNN models that can be expressed by Equation (2).

Approximate Graph Propagation (AGP). In general, it is computational infeasible to compute Equation (2) exactly as the summation goes to infinity. Following [8, 35], we will consider an approximate version of the graph propagation equation (2):

Definition 1.1 (Approximate propagation with relative error). *Let $\boldsymbol{\pi}$ be the graph propagation vector defined in Equation* (2). *Given an error threshold δ, an approximate propagation algorithm has to return an estimation vector $\hat{\boldsymbol{\pi}}$, such that for any $v \in V$, with $|\boldsymbol{\pi}(v)| > \delta$, we have*

$$|\boldsymbol{\pi}(v) - \hat{\boldsymbol{\pi}}(v)| \le \frac{1}{10} \cdot \boldsymbol{\pi}(v)$$

with at least a constant probability (e.g. 99%).

We note that some previous works [37, 40] consider the guarantee $|\boldsymbol{\pi}(v) - \hat{\boldsymbol{\pi}}(v)| \le \varepsilon_r \cdot \boldsymbol{\pi}(v)$ with probability at least $1 - p_f$, where ε_r is the relative error parameter and p_f is the fail probability. However, ε_r and p_f are set to be constant in these works. For sake of simplicity and readability, we set $\varepsilon_r = 1/10$ and $p_f = 1\%$ and introduce only one error parameter δ, following the setting in [8, 35].

Motivations. Existing works on proximity computation and GNN feature propagation are on a case-by-case basis, with each paper focusing on a particular proximity measure. For example, despite the similarity between Personalized PageRank and heat kernel PageRank, the two proximity measures admit two completely different sets of algorithms (see [14, 22, 33, 36, 38] for Personalized PageRank and [12, 13, 27, 40] for heat kernel PageRank). Therefore, a natural question is

Is there a universal algorithm that computes the approximate graph propagation with near optimal cost?

Contributions. In this paper, we present AGP, a UNIFIED randomized algorithm that computes Equation (2) with almost optimal computation time and theoretical error guarantee. AGP naturally generalizes to various proximity measures, including transition probabilities, PageRank and Personalized PageRank, heat kernel PageRank, and Katz. We conduct an extensive experimental study to demonstrate the effectiveness of AGP on real-world graphs. We show that AGP outperforms the state-of-the-art methods for local graph clustering with heat kernel PageRank. We also show that AGP can scale various GNN models (including SGC [39], APPNP [25], and GDC [26]) up on the billion-edge graph Papers100M, which is the largest publicly available GNN dataset.

2 PRELIMINARY AND RELATED WORK

In this section, we provide a detailed discussion on how the graph propagation equation (2) models various node proximity measures. Table 2 summarizes the notations used in this paper.

Personalized PageRank (PPR) [32] is developed by Google to rank web pages on the world wide web, with the intuition that "a page is important if it is referenced by many pages, or important pages". Given an undirected graph $G = (V, E)$ with n nodes and m edges and a *teleporting probability distribution* $\boldsymbol{x}$ over the n nodes, the PPR vector $\boldsymbol{\pi}$ is the solution to the following equation:

$$\boldsymbol{\pi} = (1-\alpha) \cdot \mathbf{A}\mathbf{D}^{-1} \cdot \boldsymbol{\pi} + \alpha \boldsymbol{x}. \tag{3}$$

The unique solution to Equation (3) is given by $\boldsymbol{\pi} = \sum_{i=0}^{\infty} \alpha (1-\alpha)^i \cdot \left(\mathbf{A}\mathbf{D}^{-1}\right)^i \cdot \boldsymbol{x}$.

The efficient computation of the PPR vector has been extensively studied for the past decades. A simple algorithm to estimate PPR is the Monte-Carlo sampling [15], which stimulates adequate random walks from the source node s generated following $\boldsymbol{x}$ and uses the percentage of the walks that terminate at node v as the estimation of $\boldsymbol{\pi}(v)$. Forward Search [5] conducts deterministic local pushes from the source node s to find nodes with large PPR scores. FORA [37] combines Forward Search with the Monte Carlo method to improve the computation efficiency. TopPPR [38] combines Forward Search, Monte Carlo, and Backward Search [30] to obtain a better error guarantee for top-k PPR estimation. ResAcc [29] refines FORA by accumulating probability masses before each push. However, these methods only work for graph propagation with transition matrix $\mathbf{A}\mathbf{D}^{-1}$. For example, the Monte Carlo method simulates random walks to obtain the estimation, which is not possible with the general propagation matrix $\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}$.

Another line of research [30, 35] studies the *single-target PPR*, which asks for the PPR value of every node to a given target node v on the graph. The single-target PPR vector for a given node v is defined by the slightly different formula:

$$\boldsymbol{\pi} = (1-\alpha) \cdot \left(\mathbf{D}^{-1}\mathbf{A}\right) \cdot \boldsymbol{\pi} + \alpha \boldsymbol{e}_v. \tag{4}$$

Unlike the single-source PPR vector, the single-target PPR vector $\boldsymbol{\pi}$ is not a probability distribution, which means $\sum_{s \in V} \boldsymbol{\pi}(s)$ may not

Table 1: Typical graph propagation equations.

	Algorithm	a	b	w_i	x	Propagation equation
Proximity	L-hop transition probability	0	1	$w_i = 0 (i \neq L), w_L = 1$	one-hot vector $\boldsymbol{e}_s$	$\boldsymbol{\pi} = \left(\mathbf{AD}^{-1}\right)^L \cdot \boldsymbol{e}_s$
	PageRank [32]	0	1	$\alpha\left(1-\alpha\right)^i$	$\frac{1}{n} \cdot \mathbf{1}$	$\boldsymbol{\pi} = \sum_{i=0}^{\infty} \alpha\left(1-\alpha\right)^i \cdot \left(\mathbf{AD}^{-1}\right)^i \cdot \frac{1}{n}$
	Personalized PageRank [32]	0	1	$\alpha\left(1-\alpha\right)^i$	teleport probability distribution x	$\boldsymbol{\pi} = \sum_{i=0}^{\infty} \alpha\left(1-\alpha\right)^i \cdot \left(\mathbf{AD}^{-1}\right)^i \cdot \boldsymbol{x}$
	single-target PPR [30]	1	0	$\alpha\left(1-\alpha\right)^i$	one-hot vector $\boldsymbol{e}_v$	$\boldsymbol{\pi} = \sum_{i=0}^{\infty} \alpha\left(1-\alpha\right)^i \cdot \left(\mathbf{D}^{-1}\mathbf{A}\right)^i \cdot \boldsymbol{e}_v$
	heat kernel PageRank [12]	0	1	$e^{-t} \cdot \frac{t^i}{i!}$	one-hot vector $\boldsymbol{e}_s$	$\boldsymbol{\pi} = \sum_{i=0}^{\infty} e^{-t} \cdot \frac{t^i}{i!} \cdot \left(\mathbf{AD}^{-1}\right)^i \cdot \boldsymbol{e}_s$
	Katz [23]	0	0	β^i	one-hot vector $\boldsymbol{e}_s$	$\boldsymbol{\pi} = \sum_{i=0}^{\infty} \beta^i \mathbf{A}^i \cdot \boldsymbol{e}_s$
GNN	SGC [39]	$\frac{1}{2}$	$\frac{1}{2}$	$w_i = 0 (i \neq L), w_L = 1$	the graph signal $\boldsymbol{x}$	$\boldsymbol{\pi} = \left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\right)^L \cdot \boldsymbol{x}$
	APPNP [25]	$\frac{1}{2}$	$\frac{1}{2}$	$\alpha\left(1-\alpha\right)^i$	the graph signal $\boldsymbol{x}$	$\boldsymbol{\pi} = \sum_{i=0}^{L} \alpha\left(1-\alpha\right)^i \cdot \left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\right)^i \cdot \boldsymbol{x}$
	GDC [26]	$\frac{1}{2}$	$\frac{1}{2}$	$e^{-t} \cdot \frac{t^i}{i!}$	the graph signal $\boldsymbol{x}$	$\boldsymbol{\pi} = \sum_{i=0}^{L} e^{-t} \cdot \frac{t^i}{i!} \cdot \left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\right)^i \cdot \boldsymbol{x}$

Table 2: Table of notations.

Notation	Description
$G = (V, E)$	undirected graph with vertex and edge sets V and E
n, m	the numbers of nodes and edges in G
$\mathbf{A}, \mathbf{D}$	the adjacency matrix and degree matrix of G
N_u, d_u	the neighbor set and the degree of node u
a, b	the Laplacian parameters
$\boldsymbol{x}$	the graph signal vector in $\mathcal{R}^n$, $\lVert\boldsymbol{x}\rVert_2 = 1$
w_i, Y_i	the i-th weight and partial sum $Y_i = \sum_{k=i}^{\infty} w_k$
$\boldsymbol{\pi}, \hat{\boldsymbol{\pi}}$	the true and estimated propagation vectors in $\mathcal{R}^n$
$\boldsymbol{r}^{(i)}, \hat{\boldsymbol{r}}^{(i)}$	the true and estimated i-hop residue vectors in $\mathcal{R}^n$
$\boldsymbol{q}^{(i)}, \hat{\boldsymbol{q}}^{(i)}$	the true and estimated i-hop reserve vectors in $\mathcal{R}^n$
δ	the relative error threshold
$\tilde{O}$	the Big-Oh natation ignoring the log factors

equal to 1. We can also derive the unique solution to Equation (4) by $\boldsymbol{\pi} = \sum_{i=0}^{\infty} \alpha\left(1-\alpha\right)^i \cdot \left(\mathbf{D}^{-1}\mathbf{A}\right)^i \cdot \boldsymbol{e}_v$.

Heat kernal PageRank is proposed by [12] for high quality community detection. For each node $v \in V$ and the seed node s, the heat kernel PageRank (HKPR) $\boldsymbol{\pi}(v)$ equals to the probability that a heat kernal random walk starting from node s ends at node v. The length L of the random walks follows the Poisson distribution with parameter t, i.e. $\Pr[L = i] = \frac{e^{-t}t^i}{i!}, i = 0, \ldots, \infty$. Consequently, the HKPR vector of a given node s is defined as $\boldsymbol{\pi} = \sum_{i=0}^{\infty} \frac{e^{-t}t^i}{i!} \cdot \left(\mathbf{AD}^{-1}\right)^i \cdot \boldsymbol{e}_s$, where $\boldsymbol{e}_s$ is the one-hot vector with $\boldsymbol{e}_s(s) = 1$. This equation fits in the framework of our generalized propagation equation (2) if we set $a = 0, b = 1, w_i = \frac{e^{-t}t^i}{i!}$, and $\boldsymbol{x} = \boldsymbol{e}_s$. Similar to PPR, HKPR can be estimated by the Monte Carlo method [13, 40] that simulates random walks of Possion distributed length. HK-Relax [27] utilizes Forward Search to approximate the HKPR vector. TEA [40] combines Forward Search with Monte Carlo for a more accurate estimator.

Katz index [23] is another popular proximity measurement to evaluate relative importance of nodes on the graph. Given two node s and v, the Katz score between s and v is characterized by the number of reachable paths from s to v. Thus, the Katz vector for a given source node s can be expressed as $\boldsymbol{\pi} = \sum_{i=0}^{\infty} \mathbf{A}^i \cdot \boldsymbol{e}_s$, where $\mathbf{A}$ is the adjacency matrix and $\boldsymbol{e}_s$ is the one-hot vector with $\boldsymbol{e}_s(s)$=1. However, this summation may not converge due to the large spectral span of $\mathbf{A}$. A commonly used fix up is to apply a penalty of β to each step of the path, leading to the following definition: $\boldsymbol{\pi} = \sum_{i=0}^{\infty} \beta^i \cdot \mathbf{A}^i \cdot \boldsymbol{e}_s$. To guarantee convergence, β is a constant that set to be smaller than $\frac{1}{\lambda_1}$, where λ_1 is the largest eigenvalue of the adjacency matrix $\mathbf{A}$. Similar to PPR, the Katz vector can be computed by iterative multiplying $\boldsymbol{e}_s$ with $\mathbf{A}$, which runs in $\tilde{O}(m+n)$ time [16]. Katz has been widely used in graph analytic and learning tasks such as link prediction [28] and graph embedding [31]. However, the $\tilde{O}(m+n)$ computation time limits its scalability on large graphs.

Proximity-based Graph Neural Networks. Consider an undirected graph $G = (V, E)$, where V and E represent the set of vertices and edges. Each node v is associated with a numeric feature vector of dimension d. The n feature vectors form an $n \times d$ matrix $\mathbf{X}$. Following the convention of graph neural networks [18, 24], we assume each node in G is also attached with a self-loop. The goal of graph neural network is to obtain an $n \times d'$ representation matrix $\mathbf{Z}$, which encodes both the graph structural information and the feature matrix $\mathbf{X}$. Kipf and Welling [24] propose the vanilla Graph Convolutional Network (GCN), of which the ℓ-th representation $\mathbf{H}^{(\ell)}$ is defined as

$$\mathbf{H}^{(\ell)} = \sigma\left(\mathbf{D}^{-\frac{1}{2}} \cdot \mathbf{A} \cdot \mathbf{D}^{-\frac{1}{2}} \cdot \mathbf{H}^{(\ell-1)} \cdot \mathbf{W}^{(\ell)}\right), \tag{5}$$

where $\mathbf{A}$ and $\mathbf{D}$ are the adjacency matrix and the diagonal degree matrix of G, $\mathbf{W}^{(\ell)}$ is the learnable weight matrix, and $\sigma(.)$ is a nonlinear activation function (a common choice is the Relu function). Let L denote the number of layers in the GCN model. The 0-th representation $\mathbf{H}^{(0)}$ is set to the feature matrix $\mathbf{X}$, and the final representation matrix $\mathbf{Z}$ is the L-th representation $\mathbf{H}^{(L)}$. Intuitively, GCN aggregates the neighbors' representation vectors from the $(\ell-1)$-th layer to form the representation of the ℓ-th layer. Such a simple paradigm is proved to be effective in various graph learning tasks [18, 24].

A major drawback of the vanilla GCN is the lack of ability to scale on graphs with millions of nodes. Such limitation is caused by the fact that the vanilla GCN uses a full-batch training process and stores each node's representation in the GPU memory. To extend GNN to large graphs, a line of research focuses on decoupling prediction and propagation, which removes the non-linear activation function $\sigma(.)$ for better scalability. These methods first apply a proximity matrix $\mathbf{P}$ to the feature matrix $\mathbf{X}$ to obtain the representation matrix $\mathbf{Z}$, and then feed $\mathbf{Z}$ into logistic regression or standard neural network for predictions. Among them, SGC [39] simplifies the vanilla GCN by taking the multiplication of the L-th normalized transition probability matrix $\mathbf{P} = \left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\right)^L$ and feature matrix $\mathbf{X}$ to form the final presentation $\mathbf{Z} = \mathbf{P} \cdot \mathbf{X} = \left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\right)^L \cdot \mathbf{X}$. The proximity matrix $\mathbf{P}$ can be generalized to PPR used in APPNP [25] and HKPR used in GDC [26]. Note that the original APPNP [25] first applies an one-layer neural network to $\mathbf{X}$ before the propagation that $\mathbf{Z}^{(k+1)} = f_\theta(\mathbf{X})$. Then APPNP propagates the feature matrix $\mathbf{X}$ with a truncated Personalized PageRank matrix $\mathbf{Z} = \sum_{i=0}^{L} \alpha (1-\alpha)^i \cdot \left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\right)^i \cdot f_\theta(\mathbf{X})$, where α is a constant in $(0,1)$. The original GDC [26] follows the structure of GCN and substitutes the diffusion kernel to heat kernel. For the sake of high scalability, unless specified otherwise, we use APPNP and GDC to denote the linear propagation process that $\mathbf{Z} = \sigma\left(\sum_{i=0}^{L} \mathbf{P}^{(i)} \cdot \mathbf{X}\right)$ throughout the paper, where L is the number of layers. Specifically, $\mathbf{P}^{(i)} = \alpha (1-\alpha)^i \cdot \left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\right)^i$ for APPNP and $\mathbf{P}^{(i)} = \frac{e^{-t}t^i}{i!} \cdot \left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\right)^i$ for GDC.

A recent work PPRGo [7] improves the scalability of APPNP by employing the Forward Search algorithm [5] to perform the propagation. However, PPRGo only works for APPNP, which, as we shall see in our experiment, may not always achieve the best performance among the three models. Finally, a recent work GBP [10] proposes to use deterministic local push and the Monte Carlo method to approximate GNN propagation of the form $\mathbf{Z} = \sum_{i=0}^{L} w_i \cdot \left(\mathbf{D}^{-(1-r)}\mathbf{A}\mathbf{D}^{-r}\right)^i \cdot \mathbf{X}$. However, GBP suffers from two drawbacks: 1) it requires $a+b=1$ in Equation (2) to utilize the Monte-Carlo method, and 2) it requires $O(n)$ space cost in the Monte-Carlo process, which limits its scalability on large graphs with billions of edges. As we shall see in Section 5, to run GBP on Friendster and Papers100M, we have to reduce the forward Monte-Carlo phase and only conduct the reverse push to avoid the out-of-memory problems.

3 BASIC PROPAGATION

In the next two sections, we present two algorithms to compute the graph propagation equation (2) with the theoretical relative error guarantee in Definition 1.1.

Assumption on graph signal $\boldsymbol{x}$. For sake of simplicity, we assume the graph signal $\boldsymbol{x}$ is non-negative. We can deal with the negative entries in $\boldsymbol{x}$ by decomposing it into $\boldsymbol{x} = \boldsymbol{x}^+ + \boldsymbol{x}^-$, where $\boldsymbol{x}^+$ only contains the non-negative entries of $\boldsymbol{x}$ and $\boldsymbol{x}^-$ only contains the negative entries of $\boldsymbol{x}$. After we compute $\boldsymbol{\pi}^+ = \sum_{i=0}^{\infty} w_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \boldsymbol{x}^+$ and $\boldsymbol{\pi}^- = \sum_{i=0}^{\infty} w_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \boldsymbol{x}^-$, we can combine $\boldsymbol{\pi}^+$ and $\boldsymbol{\pi}^-$ to form $\boldsymbol{\pi} = \boldsymbol{\pi}^+ + \boldsymbol{\pi}^-$. We will also assume $\boldsymbol{x}$ is normalized, that is $\|\boldsymbol{x}\|_2 = 1$.

Assumptions on w_i. To make the computation of Equation (2) feasible, we first introduce several assumptions on the weight sequence w_i for $i \in \{0, 1, 2, ...\}$. We assume $\sum_{i=0}^{\infty} w_i = 1$. If not, we can perform propagation with $w_i' = w_i / \sum_{i=0}^{\infty} w_i$ and rescale the result by $\sum_{i=0}^{\infty} w_i$. We also note that to ensure the convergence of Equation 2, the weight sequence w_i has to satisfy $\sum_{i=0}^{\infty} w_i \lambda_{max}^i < \infty$, where λ_{max} is the maximum singular value of the propagation matrix $\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}$. Therefore, we assume that for sufficiently large i, $w_i \lambda_{max}^i$ is upper bounded by a geometric distribution:

Assumption 3.1. *There exists a constant L_0 and $\lambda < 1$, such that for any $i \geq L_0$, $w_i \cdot \lambda_{max}^i \leq \lambda^i$.*

According to Assumption 3.1, to achieve the relative error in Definition 1.1, we only need to compute the prefix sum $\boldsymbol{\pi} = \sum_{i=0}^{L} w_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \boldsymbol{x}$, where L equals to $\log_\lambda \delta = O\left(\log \frac{1}{\delta}\right)$. This property is possessed by all proximity measures discussed in this paper. For example, PageRank and Personalized PageRank set $w_i = \alpha (1-\alpha)^i$, where α is a constant. Since the maximum eigenvalue of $\mathbf{A}\mathbf{D}^{-1}$ is 1, we have $\| \sum_{i=L+1}^{\infty} w_i \mathbf{A}\mathbf{D}^{-1} \cdot \boldsymbol{x}\|_2 \leq \| \sum_{i=L+1}^{\infty} w_i \cdot \boldsymbol{x}\|_2 \leq \sum_{i=L+1}^{\infty} \alpha \cdot (1-\alpha)^i = (1-\alpha)^{L+1}$. In the last inequality, we use the assumption on $\boldsymbol{x}$ that $\|\boldsymbol{x}\|_2 = 1$. If we set $L = \log_{1-\alpha} \delta = O\left(\log \frac{1}{\delta}\right)$, the remaining sum $\| \sum_{i=L+1}^{\infty} w_i \mathbf{A}\mathbf{D}^{-1} \cdot \boldsymbol{x}\|_2$ is bounded by δ. We can prove similar bounds for HKPR, Katz, and transition probability as well.

Basic Propagation. As a baseline solution, we can compute the graph propagation equation (2) by iteratively updating the propagation vector $\boldsymbol{\pi}$ via matrix-vector multiplications. Similar approaches have been used for computing PageRank, PPR, HKPR and Katz, under the names of Power Iteration or Power Method.

In general, we employ matrix-vector multiplications to compute the summation of the first $L = O\left(\log \frac{1}{\delta}\right)$ hops of equation (2): $\boldsymbol{\pi} = \sum_{i=0}^{L} w_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \boldsymbol{x}$. To avoid the $O(nL)$ space of storing vectors $\left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \boldsymbol{x}, i = 0, \ldots, L$, we only use two vectors: the *residue and reserve vectors*, which are defined as follows.

Definition 3.1. *[**residue and reserve**] Let $Y_i = \sum_{k=i}^{\infty} w_k, i = 0, \ldots, \infty$, denote the partial sum of the weight sequence. Recall that $Y_0 = \sum_{k=0}^{\infty} w_k = 1$. At level i, the residue vector is defined as $\boldsymbol{r}^{(i)} = Y_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \boldsymbol{x}$; The reserve vector is defined as $\boldsymbol{q}^{(i)} = \frac{w_i}{Y_i} \cdot \boldsymbol{r}^{(i)} = w_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \boldsymbol{x}$.*

Intuitively, for each node $u \in V$ and level $i \geq 0$, the residue $\boldsymbol{r}^{(i)}(u)$ denotes the probability mass to be distributed to node u at level i, and the reserve $\boldsymbol{q}^{(i)}(u)$ denotes the probability mass that will stay at node u in level i permanently. By Definition 3.1, the graph propagation equation (2) can be expressed as $\boldsymbol{\pi} = \sum_{i=0}^{\infty} \boldsymbol{q}^{(i)}$. Furthermore, the residue vector $\boldsymbol{r}^{(i)}$ satisfies the following recursive formula:

$$\boldsymbol{r}^{(i+1)} = \frac{Y_{i+1}}{Y_i} \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right) \cdot \boldsymbol{r}^{(i)}. \tag{6}$$

We also observe that the reserve vector $\boldsymbol{q}^{(i)}$ can be derived from the residue vector $\boldsymbol{r}^{(i)}$ by $\boldsymbol{q}^{(i)} = \frac{w_i}{Y_i} \cdot \boldsymbol{r}^{(i)}$. Consequently, given a predetermined level number L, we can compute the graph propagation equation (2) by iteratively computing the residue vector $\boldsymbol{r}^{(i)}$ and reserve vector $\boldsymbol{q}^{(i)}$ for $i = 0, 1, ..., L$.

Algorithm 1: Basic Propagation Algorithm

```
Input: Undirected graph $G = (V, E)$, graph signal vector $\boldsymbol{x}$,
       weights $w_i$, number of levels $L$
Output: the estimated propagation vector $\hat{\boldsymbol{\pi}}$
1 $\boldsymbol{r}^{(0)} \leftarrow \boldsymbol{x}$;
2 for $i = 0$ to $L-1$ do
3     for each $u \in V$ with nonzero $\boldsymbol{r}^{(i)}(u)$ do
4         for each $v \in N_u$ do
5             $\boldsymbol{r}^{(i+1)}(v) \leftarrow \boldsymbol{r}^{(i+1)}(v) + \left(\frac{Y_{i+1}}{Y_i}\right) \cdot \frac{\boldsymbol{r}^{(i)}(u)}{d_v^a \cdot d_u^b}$;
6         $\boldsymbol{q}^{(i)}(u) \leftarrow \boldsymbol{q}^{(i)}(u) + \frac{w_i}{Y_i} \cdot \boldsymbol{r}^{(i)}(u)$;
7     $\hat{\boldsymbol{\pi}} \leftarrow \hat{\boldsymbol{\pi}} + \boldsymbol{q}^{(i)}$ and empty $\boldsymbol{r}^{(i)}, \boldsymbol{q}^{(i)}$;
8 $\boldsymbol{q}^{(L)} = \frac{w_L}{Y_L} \cdot \boldsymbol{r}^{(L)}$ and $\hat{\boldsymbol{\pi}} \leftarrow \hat{\boldsymbol{\pi}} + \boldsymbol{q}^{(L)}$;
9 return $\hat{\boldsymbol{\pi}}$;
```

Algorithm 1 illustrates the pseudo-code of the basic iterative propagation algorithm. We first set $\boldsymbol{r}^{(0)} = \boldsymbol{x}$ (line 1). For i from 0 to $L-1$, we compute $\boldsymbol{r}^{(i+1)} = \frac{Y_{i+1}}{Y_i} \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right) \cdot \boldsymbol{r}^{(i)}$ by pushing the probability mass $\left(\frac{Y_{i+1}}{Y_i}\right) \cdot \frac{\boldsymbol{r}^{(i)}(u)}{d_v^a \cdot d_u^b}$ to each neighbor v of each node u (lines 2-5). Then, we set $\boldsymbol{q}^{(i)} = \frac{w_i}{Y_i} \cdot \boldsymbol{r}^{(i)}$ (line 6), and aggregate $\boldsymbol{q}^{(i)}$ to $\hat{\boldsymbol{\pi}}$ (line 7). We also empty $\boldsymbol{r}^{(i)}, \boldsymbol{q}^{(i)}$ to save memory. After all L levels are processed, we transform the residue of level L to the reserve vector by updating $\hat{\boldsymbol{\pi}}$ accordingly (line 8). We return $\hat{\boldsymbol{\pi}}$ as an estimator for the graph propagation vector $\boldsymbol{\pi}$ (line 9).

Intuitively, each iteration of Algorithm 1 computes the matrix-vector multiplication $\boldsymbol{r}^{(i+1)} = \frac{Y_{i+1}}{Y_i} \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right) \cdot \boldsymbol{r}^{(i)}$, where $\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}$ is an $n \times n$ sparse matrix with m non-zero entries. Therefore, the cost of each iteration of Algorithm 1 is $O(m)$. To achieve the relative error guarantee in Definition 1.1, we need to set $L = O\left(\log \frac{1}{\delta}\right)$, and thus the total cost becomes $O\left(m \cdot \log \frac{1}{\delta}\right)$. Due to the logarithmic dependence on δ, we use Algorithm 1 to compute high-precision proximity vectors as the ground truths in our experiments. However, the linear dependence on the number of edges m limits the scalability of Algorithm 1 on large graphs. In particular, in the setting of Graph Neural Network, we treat each column of the feature matrix $\mathbf{X} \in \mathcal{R}^{n \times d}$ as the graph signal $\boldsymbol{x}$ to do the propagation. Therefore, Algorithm 1 costs $O\left(md \log \frac{1}{\delta}\right)$ to compute the representation matrix $\mathbf{Z}$. Such high complexity limits the scalability of the existing GNN models.

4 RANDOMIZED PROPAGATION

A failed attempt: pruned propagation. The $O\left(m \log \frac{1}{\delta}\right)$ running time is undesirable in many applications. To improve the scalability of the basic propagation algorithm, a simple idea is to prune the nodes with small residues in each iteration. This approach has been widely adapted in local clustering methods such as Nibble and PageRank-Nibble [5]. In general, there are two schemes to prune the nodes: 1) we can ignore a node u if its residue $\hat{\boldsymbol{r}}^{(i)}(u)$ is smaller than some threshold ε in line 3 of Algorithm 1, or 2) in line 4 of Algorithm 1, we can somehow ignore an edge (u, v) if $\left(\frac{Y_{i+1}}{Y_i}\right) \cdot \frac{\boldsymbol{r}^{(i)}(u)}{d_v^a \cdot d_u^b}$, the

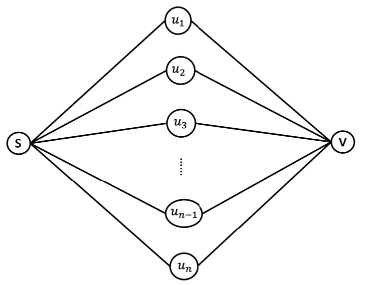

Figure 1: A bad-case graph for pruned propagation.

residue to be propagated from u to v, is smaller than some threshold ε'. Intuitively, both pruning schemes can reduce the number of operations in each iteration.

However, as it turns out, the two approaches suffer from either unbounded error or large time cost. More specifically, consider the toy graph shown in Figure 1, on which the goal is to estimate $\boldsymbol{\pi} = \left(\mathbf{A}\mathbf{D}^{-1}\right)^2 \cdot \boldsymbol{e}_s$, the transition probability vector of a 2-step random walk from node s. It is easy to see that $\boldsymbol{\pi}(v) = 1/2$, $\boldsymbol{\pi}(s) = 1/2$, and $\boldsymbol{\pi}(u_i) = 0, i = 1, \ldots, n$. We focus on constant relative error threshold of $\boldsymbol{\pi}(v)$'s approximation (e.g. $\delta = 1/4$). Hence, the approximate propagation algorithm has to return a constant approximation of $\boldsymbol{\pi}(v)$. We consider the first iteration, which pushes the residue $\boldsymbol{r}^{(0)}(s) = 1$ to $u_1, \ldots, u_n$. If we adopt the first pruning scheme that performs push on s when the residue is large, then we will have to visit all n neighbors of s, leading to an intolerable time cost of $O(n)$. On the other hand, we observe that the residue transforms from s to any neighbor u_i is $\frac{\boldsymbol{r}^{(0)}(s)}{d_s} = \frac{1}{n}$. Therefore, if we adopt the second pruning scheme which only performs pushes on edges with large residues to be transformed to, we will simply ignore all pushes from s to $u_1, \ldots, u_n$ and make the incorrect estimation that $\hat{\boldsymbol{\pi}}(v) = 0$. The problem becomes worse when we are dealing with the general graph propagation equation (2), where the Laplacian parameters a and b in the transition matrix $\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}$ may take arbitrary values. For example, to the best of our knowledge, no sub-linear approximate algorithm exists for Katz index where $a = b = 0$.

Randomized propagation. We solve the above dilemma by presenting a simple *randomized* propagation algorithm that achieves both theoretical approximation guarantee and near-optimal running time complexity. Algorithm 2 illustrates the pseudo-code of the Randomized Propagation Algorithm, which only differs from Algorithm 1 by a few lines (4-9). Similar to Algorithm 1, Algorithm 2 takes in an undirected graph $G = (V, E)$, a graph signal vector $\boldsymbol{x}$, a level number L and a weighted sequence w_i for $i \in [0, L]$. In addition, Algorithm 2 takes in an extra parameter ε, which specifies the relative error guarantee. As we shall see in the analysis, ε is roughly of the same order as the relative error threshold δ in Definition 1.1. Similar to Algorithm 1, we start with $\hat{\boldsymbol{r}}^{(0)} = \boldsymbol{x}$ and iteratively perform propagation through level 0 to level L. The key difference is that, on a node u with non-zero residue $\hat{\boldsymbol{r}}^{(i)}(u)$, instead of pushing the residue to the whole neighbor set N_u, we only perform pushes to neighbor v with small degree d_v. More specifically, for each neighbor $v \in N(u)$ with degree $d_v \le \left(\frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_u^b}\right)^{1/a}$, we increase v's residue by $\frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_v^a \cdot d_u^b}$, which is the same value as in Algorithm 1.

Algorithm 2: Randomized Propagation Algorithm

```
Input: undirected graph $G = (V, E)$, graph signal vector $\boldsymbol{x}$
       with $\|\boldsymbol{x}\|_1 \le 1$, weighted sequence $w_i (i = 0, 1, ..., L)$,
       error parameter $\varepsilon$, number of levels $L$
Output: the estimated propagation vector $\hat{\boldsymbol{\pi}}$
1  $\hat{\boldsymbol{r}}^{(0)} \leftarrow \boldsymbol{x}$;
2  for $i = 0$ to $L-1$ do
3      for each $u \in V$ with non-zero residue $\hat{\boldsymbol{r}}^{(i)}(u)$ do
4          for each $v \in N_u$ and $d_v \le \left(\frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_u^b}\right)^{\frac{1}{a}}$ do
5              $\hat{\boldsymbol{r}}^{(i+1)}(v) \leftarrow \hat{\boldsymbol{r}}^{(i+1)}(v) + \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_v^a \cdot d_u^b}$;
6          Subset Sampling: Sample each remaining neighbor
           $v \in N_u$ with probability $p_v = \frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_u^b} \cdot \frac{1}{d_v^a}$;
7          for each sampled neighbor $v \in N(u)$ do
8              $\hat{\boldsymbol{r}}^{(i+1)}(v) \leftarrow \hat{\boldsymbol{r}}^{(i+1)}(v) + \varepsilon$;
9          $\hat{\boldsymbol{q}}^{(i)}(u) \leftarrow \hat{\boldsymbol{q}}^{(i)}(u) + \frac{w_i}{Y_i} \cdot \hat{\boldsymbol{r}}^{(i)}(u)$;
10     $\hat{\boldsymbol{\pi}} \leftarrow \hat{\boldsymbol{\pi}} + \hat{\boldsymbol{q}}^{(i)}$ and empty $\hat{\boldsymbol{r}}^{(i)}, \hat{\boldsymbol{q}}^{(i)}$;
11 $\boldsymbol{q}^{(L)} = \frac{w_L}{Y_L} \cdot \boldsymbol{r}^{(L)}$ and $\hat{\boldsymbol{\pi}} \leftarrow \hat{\boldsymbol{\pi}} + \boldsymbol{q}^{(L)}$;
12 return $\hat{\boldsymbol{\pi}}$;
```

We also note that the condition $d_v \le \left(\frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_u^b}\right)^{1/a}$ is equivalent to $\frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_v^a \cdot d_u^b} > \varepsilon$, which means we push the residue from u to v only if it is larger than ε. For the remaining nodes in N_u, we sample each neighbor $v \in N_u$ with probability $p_v = \frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_v^a \cdot d_u^b}$. Once a node v is sampled, we increase the residue of v by ε. The choice of p_v is to ensure that $p_v \cdot \varepsilon$, the expected residue increment of v, equals to $\frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_v^a \cdot d_u^b}$, the true residue increment if we perform the actual propagation from u to v in Algorithm 1.

There are two key operations in Algorithm 2. First of all, we need to access the neighbors with small degrees. Secondly, we need to sample each (remaining) neighbor $v \in N_u$ according to some probability p_v. Both operations can be supported by scanning over the neighbor set N_u. However, the cost of the scan is asymptotically the same as performing a full propagation on u (lines 4-5 in Algorithm 1), which means Algorithm 2 will lose the benefit of randomization and essentially become the same as Algorithm 1.

Pre-sorting adjacency list by degrees. To access the neighbors with small degrees, we can pre-sort each adjacency list N_u according to the degrees. More precisely,we assume that $N_u = \{v_1, \ldots, v_{d_u}\}$ is stored in a way that $d_{v_1} \le \ldots \le d_{v_{d_u}}$. Consequently, we can implement lines 4-5 in Algorithm 2 by sequentially scanning through $N_u = \{v_1, \ldots, v_{d_u}\}$ and stopping at the first v_j such that $d_{v_j} > \left(\frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_u^b}\right)^{1/a}$. With this implementation, we only need to access the neighbors with degrees that exceed the threshold. We also note that we can pre-sort the adjacency lists when reading the graph into the memory, without increasing the asymptotic cost. In particular, we construct a tuple (u, v, d_v) for each edge (u, v) and use counting sort to sort (u, v, d_v) tuples in the ascending order of d_v. Then we scan the tuple list. For each (u, v, d_v), we append v to the end of u's adjacency list N_u. Since each d_v is bounded by n, and there are m tuples, the cost of counting sort is bounded by $O(m+n)$, which is asymptotically the same as reading the graphs.

Subset Sampling. The second problem, however, requires a more delicate solution. Recall that the goal is to sample each neighbor $v_j \in N_u = \{v_1, \ldots, v_{d_u}\}$ according to probability $p_j = \frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_u^b} \cdot \frac{1}{d_{v_j}^a}$ without touching all the neighbors in N_u. This problem is known as the *Subset Sampling problem* and has been solved optimally in [9]. For ease of implementation, we employ a simplified solution: we partition the adjacency list $v_j \in N_u = \{v_1, \ldots, v_{d_u}\}$ into $O(\log n)$ groups, such that the k-th group G_k consists of neighbors v_j with degrees $2^k \le d_{v_j} \le 2^{k+1} - 1$. Note that this can be done by simply sorting N_u according to the degrees. Inside the k-th group G_k, the sampling probability $p_j = \frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_u^b} \cdot \frac{1}{d_{v_j}^a}$ differs by a factor of at most $2^a \le 2$. Let p^* denote the maximum sampling probability in G_k, we generate a random integer ℓ according to the Binomial distribution $B(|G_k|, p^*)$, and randomly selected ℓ neighbors from G_k. For each selected neighbor v_j, we reject it with probability $1 - p_j/p^*$. Note that the sampling complexity for G_k is $O\left(\sum_{j \in G_k} p_j + 1\right)$. Consequently, the total sampling complexity becomes $O\left(\sum_{k=1}^{\log n}\left(\sum_{j \in G_k} p_j + 1\right)\right) = O\left(\sum_{j=0}^{d_u} p_j + \log n\right)$. Note that for each subset sampling operation, we need to return $O\left(\sum_{j=0}^{d_u} p_j\right)$ neighbors in expectation, so this complexity is optimal up to the $\log n$ additive term.

Analysis. We now present a series of lemmas that characterize the error guarantee and the running time of Algorithm 2. For readability, we only give some intuitions for each Lemma and defer the detailed proofs to the technical report [1]. We first present a Lemma that shows Algorithm 2 computes unbiased estimators for the residue and reserve vectors.

Lemma 4.1. *For each node $v \in V$, Algorithm 2 computes estimators $\hat{\boldsymbol{r}}^{(\ell)}(v)$ and $\hat{\boldsymbol{q}}^{(\ell)}(v)$ such that* $\mathrm{E}\left[\hat{\boldsymbol{q}}^{(\ell)}(v)\right] = \boldsymbol{q}^{(\ell)}(v)$ *and* $\mathrm{E}\left[\hat{\boldsymbol{r}}^{(\ell)}(v)\right] = \boldsymbol{r}^{(\ell)}(v)$ *holds for* $\forall \ell \in \{0, 1, 2, ..., L\}$.

To give some intuitions on the correctness of Lemma 4.1, recall that in lines 4-5 of Algorithm 1, we add $\frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_v^a \cdot d_u^b}$ to each residue $\hat{\boldsymbol{r}}^{(i+1)}(v)$ for $\forall v \in N_u$. We perform the same operation in Algorithm 2 for each neighbor $v \in N_u$ with large degree d_v. For each remaining neighbor $v \in V$, we add a residue of ε to $\hat{\boldsymbol{r}}^{(i+1)}(v)$ with probability $\frac{1}{\varepsilon} \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_v^a \cdot d_u^b}$, leading to an expected increment of $\frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\boldsymbol{r}}^{(i)}(u)}{d_v^a \cdot d_u^b}$. Therefore, Algorithm 2 computes an unbiased estimator for each residue vector $\boldsymbol{r}^{(i)}$, and consequently an unbiased estimator for each reserve vector $\boldsymbol{q}^{(i)}$.

In the next Lemma, we bound the variance of the approximate graph propagation vector $\hat{\pi}$, which takes a surprisingly simple form.

Lemma 4.2. *For any node $v \in V$, the variance of $\hat{\boldsymbol{\pi}}(v)$ obtained by Algorithm 2 satisfies* $\mathrm{Var}\left[\hat{\boldsymbol{\pi}}(v)\right] \le \frac{L(L+1)\varepsilon}{2} \cdot \boldsymbol{\pi}(v)$.

Recall that we can set $L = O(\log 1/\varepsilon)$ to obtain a relative error threshold of ε. Lemma 4.2 essentially states that the variance

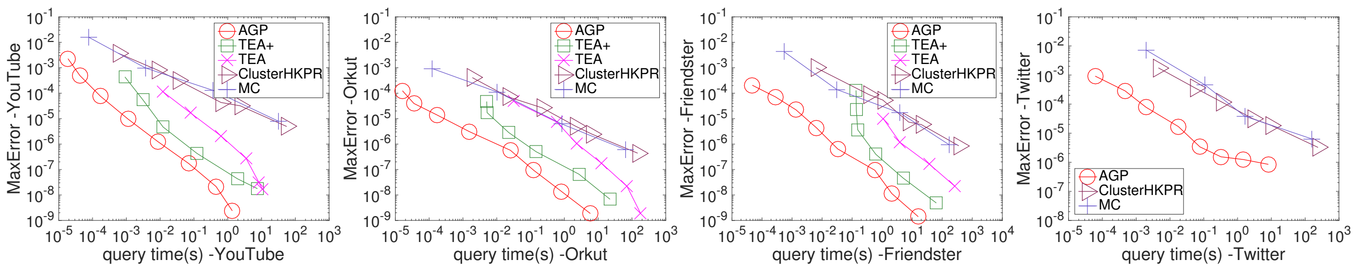

Figure 2: Tradeoffs between *MaxError* and query time in local clustering.

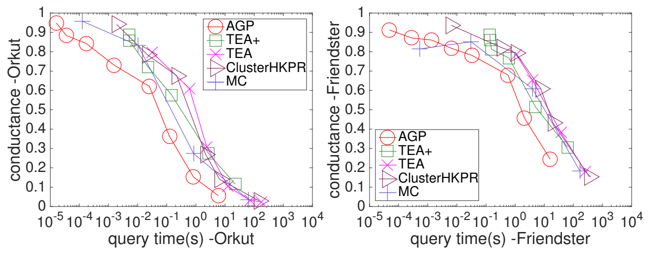

Figure 3: Tradeoffs between *conductance* and query time in local clustering.

decreases linearly with the error parameter ε. Such property is desirable for bounding the relative error. In particular, for any node v with $\boldsymbol{\pi}(v) > 100 \cdot \frac{L(L+1)\varepsilon}{2}$, the standard deviation of $\hat{\boldsymbol{\pi}}(v)$ is bounded by $\frac{1}{10}\boldsymbol{\pi}(v)$. Therefore, we can set $\varepsilon = \delta \cdot \frac{200}{L(L+1)} = \tilde{O}(\delta)$ and obtain the relative error guarantee in Definition 1.1. In particular, we have the following Theorem that bounds the expected cost of Algorithm 2 under the relative error guarantee.

Theorem 4.3. *Algorithm 2 achieves an approximate propagation with relative error δ, that is, for any node v with $\boldsymbol{\pi}(v) > \delta$, $|\boldsymbol{\pi}(v) - \hat{\boldsymbol{\pi}}(v)| \leq \frac{1}{10} \cdot \boldsymbol{\pi}(v)$. The expected time cost can be bounded by*

$$E[Cost] = O\left(\frac{L}{\delta} \cdot \sum_{i=1}^{L} \left\| Y_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \boldsymbol{x} \right\|_1 \right).$$

To understand the time complexity in Theorem 4.3, note that $\left\| \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \boldsymbol{x} \right\|_1$ is the summation of the true residues at level i. By the Pigeonhole principle, $\frac{1}{\delta}\left\| Y_i \cdot \left(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b}\right)^i \cdot \boldsymbol{x} \right\|_1$ is the upper bound of the residues at level i that are larger than δ. This bound is the output size of the propagation at level i, which means Algorithm 2 achieves near optimal time complexity.

Furthermore, we can compare the time complexity of Algorithm 2 with other state-of-the-art algorithms in specific applications. For example, in the setting of heat kernel PageRank, the goal is to estimate $\boldsymbol{\pi} = \sum_{i=0}^{\infty} e^{-t} \cdot \frac{t^i}{i!} \cdot \left(\mathbf{A}\mathbf{D}^{-1}\right)^i \cdot \boldsymbol{e}_s$ for a given node s. The state-of-the-art algorithm TEA [40] computes an approximate HKPR vector $\hat{\boldsymbol{\pi}}$ such that for any $\pi(v) > \delta$, $|\boldsymbol{\pi}(v) - \hat{\boldsymbol{\pi}}(v)| \leq \frac{1}{10} \cdot \boldsymbol{\pi}(v)$ holds for high probability. By the fact that t is the a constant and $\tilde{O}$ is the Big-Oh notation ignoring log factors, the total cost of TEA is bounded by $O\left(\frac{t \log n}{\delta}\right) = \tilde{O}\left(\frac{1}{\delta}\right)$. On the other hand, in the setting of HKPR, the time complexity of Algorithm 2 is bounded by

$$\frac{L}{\delta} \cdot \sum_{i=1}^{L} \left\| Y_i \cdot \left(\cdot \mathbf{A}^{\top}\mathbf{D}^{-1}\right)^i \cdot \boldsymbol{e}_s \right\|_1 = \frac{L}{\delta} \cdot \sum_{i=1}^{L} Y_i \leq \frac{L^2}{\delta} = \tilde{O}\left(\frac{1}{\delta}\right).$$

Table 3: Datasets for local clustering.

Data Set	Type	n	m
YouTube	undirected	1,138,499	5,980,886
Orkut-Links	undirected	3,072,441	234,369,798
Twitter	directed	41,652,230	1,468,364,884
Friendster	undirected	68,349,466	3,623,698,684

Here we use the facts that $\left\| \left(\mathbf{A}\mathbf{D}^{-1}\right)^i \cdot \boldsymbol{e}_s \right\|_1 = 1$ and $Y_i \leq 1$. This implies that under the specific application of estimating HKPR, the time complexity of the more generalized Algorithm 2 is asymptotically the same as the complexity of TEA. Similar bounds also holds for Personalized PageRank and transition probabilities. We defer detailed explanation to the technical report [1].

Propagation on directed graph. Our generalized propagation structure can also extend to directed graph by $\pi = \sum_{i=0}^{\infty} w_i \cdot \left(\mathbf{D}^{-a}\tilde{\mathbf{A}}\mathbf{D}^{-b}\right)^i \cdot \boldsymbol{x}$, where $\boldsymbol{D}$ denotes the diagonal out-degree matrix, and $\tilde{\mathbf{A}}$ represents the adjacency matrix or its transition according to specific applications. For PageRank, single-source PPR, HKPR, Katz we set $\tilde{\mathbf{A}} = \mathbf{A}^{\top}$ with the following recursive equation:

$$\boldsymbol{r}^{(i+1)}(v) = \sum_{u \in N_{in}(v)} \left(\frac{Y_{i+1}}{Y_i}\right) \cdot \frac{\boldsymbol{r}^{(i)}(u)}{d_{out}^a(v) \cdot d_{out}^b(u)}.$$

where $N_{in}(v)$ denotes the in-neighbor set of node v and $d_{out}(u)$ is the out-degree of node u.

5 EXPERIMENTS

This section experimentally evaluates AGP's performance in two concrete applications: local clustering with heat kernel PageRank and node classification with GNN. Specifically, Section 5.1 presents the experimental results of AGP in local clustering. Section 5.2 evaluates the effectiveness of AGP on existing GNN models.

5.1 Local clustering with HKPR

In this subsection, we conduct experiments to evaluate the performance of AGP in local clustering problem. We select HKPR among various node proximity measures as it achieves the state-of-the-art result for local clustering [13, 27, 40]. We will evaluate the trade-off curve between the query time and the approximation quality, as well as the trade-off curve between the query time and the clustering quality.

Datasets and Environment. We use three undirected graphs: YouTube, Orkut, and Friendster in our experiments, as most of the local clustering methods can only support undirected graphs. We also use a large directed graph Twitter to demonstrate AGP's effectiveness on directed graphs. The four datasets can be obtained from [2, 3]. We summarize the detailed information of the four datasets in Table 3.

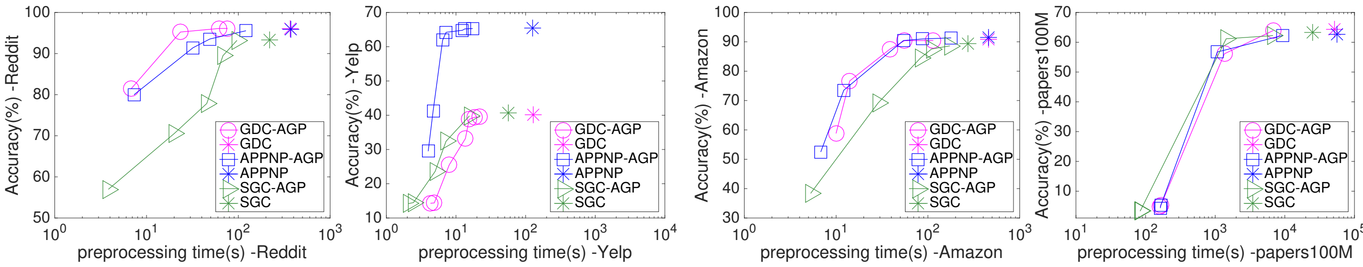

Figure 4: Tradeoffs between *Accuracy(%)* and preprocessing time in node classification (Best viewed in color).

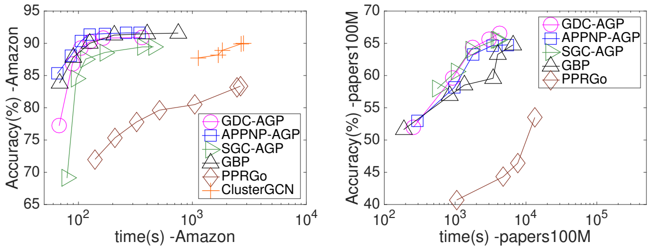

Figure 5: Comparison with GBP, PPRGo and ClusterGCN.

Methods. We compare AGP with four local clustering methods: TEA [40] and its optimized version TEA+, ClusterHKPR [13], and the Monte-Carlo method (MC). We use the results derived by the Basic Propagation Algorithm 1 with $L = 50$ as the ground truths for evaluating the trade-off curves of the approximate algorithms. Detailed descriptions on parameter setting for each method are deferred to appendix due to the space limits.

Metrics. We use *MaxError* as our metric to measure the approximation quality of each method. *MaxError* is defined as $MaxError = \max_{v\in V}\left|\frac{\boldsymbol{\pi}(v)}{d_v} - \frac{\hat{\boldsymbol{\pi}}(v)}{d_v}\right|$, which measures the maximum error between the true normalized HKPR and the estimated value. We refer to $\frac{\boldsymbol{\pi}(v)}{d_v}$ as the *normalized HKPR* value from s to v. On directed graph, d_v is substituted by the out-degree $d_{out}(v)$.

We also consider the quality of the cluster algorithms, which is measured by the *conductance*. Given a subset $S \subseteq V$, the conductance is defined as $\Phi(S)=\frac{|cut(S)|}{\min\{vol(S),2m-vol(S)\}}$, where $vol(S)=\sum_{v\in S} d(v)$, and $cut(S)=\{(u,v)\in E \mid u \in S, v \in V-S\}$. We perform a sweeping algorithm [5, 12, 13, 34, 40] to find a subset S with small conductance. More precisely, after deriving the (approximate) HKPR vector from a source node s, we sort the nodes $\{v_1,\ldots,v_n\}$ in descending order of the normalized HKPR values that $\frac{\boldsymbol{\pi}(v_1)}{d_{v_1}}\geq\frac{\boldsymbol{\pi}(v_2)}{d_{v_2}}\geq\ldots\geq\frac{\boldsymbol{\pi}(v_n)}{d_{v_n}}$. Then, we sweep through $\{v_1,\ldots,v_n\}$ and find the node set with the minimum *conductance* among partial sets $S_i=\{v_1,\ldots,v_i\}$, $i=1,\ldots,n-1$.

Experimental Results. Figure 2 plots the trade-off curve between the *MaxError* and query time. We observe that AGP achieves the lowest curve among the five algorithms on all four datasets, which means AGP incurs the least error under the same query time. As a generalized algorithm for the graph propagation problem, these results suggest that AGP outperforms the state-of-the-art HKPR algorithms in terms of the approximation quality.

To evaluate the quality of the clusters found by each method, Figure 3 shows the trade-off curve between conductance and the query time on two large undirected graphs Orkut and Friendster. We observe that AGP can achieve the lowest conductance-query time curve among the five approximate methods on both of the two datasets, which concurs with the fact that AGP provides estimators that are closest to the actual normalized HKPR.

Table 4: Datasets for node classification.

Data Set	n	m	d	Classes	Label %
Reddit	232,965	114,615,892	602	41	0.0035
Yelp	716,847	6,977,410	300	100	0.7500
Amazon	2,449,029	61,859,140	100	47	0.7000
Papers100M	111,059,956	1,615,685,872	128	172	0.0109

5.2 Node classification with GNN

In this section, we evaluate the AGP's ability to scale existing Graph Neural Network models on large graphs.

Datasets. We use four publicly available graph datasets with different size: a socal network Reddit [18], a customer interaction network Yelp [41], a co-purchasing network Amazon [11] and a large citation network Papers100M [20]. Table 4 summarizes the statistics of the datasets, where d is the dimension of the node feature, and the label rate is the percentage of labeled nodes in the graph. A detailed discussion on datasets can be found in the appendix.

GNN models. We first consider three proximity-based GNN models: APPNP [25],SGC [39], and GDC [26]. We augment the three models with the AGP Algorithm 2 to obtain three variants: APPNP-AGP, SGC-AGP and GDC-AGP. Besides, we also compare AGP with three scalable methods: PPRGo [7], GBP [10], and ClusterGCN [11].

Experimental results. Figure 4 shows the trade-off between the preprocessing time and classification accuracy for SGC, APPNP, GDC and the corresponding AGP models. For each dataset, the three snowflakes represent the exact methods SGC, APPNP, and GDC, which can be distinguished by colors. We observe that compared to the exact models, the approximate models generally achieve a 10× speedup in preprocessing time without sacrificing the classification accuracy. For example, on the billion-edge graph Papers100M, SGC-AGP achieves an accuracy of 62% in less than 2, 000 seconds, while the exact model SGC needs over 20, 000 seconds to finish.

Figure 5 shows the performances of AGP compared with PPRGo, GBP, and ClusterGCN on Amazon and Papers100M, the largest publicly available graphs for inductive and transductive node classification tasks, respectively. We present the trade-off plots between the total computation time and classification accuracy for each model. We report the total running time (i.e., preprocessing time plus training time).

We omit ClusterGCN on Papers100M as it runs out of 512GB memory on this graph. We observe that AGP significantly outperforms PPRGo and ClusterGCN on both Amazon and Papers100M in terms of both accuracy and running time. Furthermore, given the same running time, AGP achieves a higher accuracy than GBP does on Papers100M. We attribute this quality to the randomization introduced in AGP.

6 CONCLUSION

In this paper, we propose the concept of approximate graph propagation, which unifies various proximity measures, including transition probabilities, PageRank and Personalized PageRank, heat kernel PageRank, and Katz. We present a randomized graph propagation algorithm that achieves almost optimal computation time with a theoretical error guarantee. We conduct an extensive experimental study to demonstrate the effectiveness of AGP on real-world graphs. We show that AGP outperforms the state-of-the-art algorithms in the specific application of local clustering and node classification with GNNs. For future work, it is interesting to see if the AGP framework can inspire new proximity measures for graph learning and mining tasks.

7 ACKNOWLEDGEMENTS

Zhewei Wei was supported by National Natural Science Foundation of China (NSFC) No. 61972401 and No. 61932001, by the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China under Grant 18XNLG21, and by Alibaba Group through Alibaba Innovative Research Program. Hanzhi Wang was supported by the Outstanding Innovative Talents Cultivation Funded Programs 2020 of Renmin Univertity of China. Sibo Wang was supported by Hong Kong RGC ECS No. 24203419, RGC CRF No. C4158-20G, and NSFC No. U1936205. Ye Yuan was supported by NSFC No. 61932004 and No. 61622202, and by FRFCU No. N181605012. Xiaoyong Du was supported by NSFC No. 62072458. Ji-Rong Wen was supported by NSFC No. 61832017, and by Beijing Outstanding Young Scientist Program NO. BJJWZYJH012019100020098. This work was supported by Public Computing Cloud, Renmin University of China, and by China Unicom Innovation Ecological Cooperation Plan.

REFERENCES

[1] https://github.com/wanghzccls/Approximate-Graph-Propagation-Technical-Report.
[2] http://snap.stanford.edu/data.
[3] http://law.di.unimi.it/datasets.php.
[4] Reid Andersen, Christian Borgs, Jennifer Chayes, John Hopcroft, Kamal Jain, Vahab Mirrokni, and Shanghua Teng. Robust pagerank and locally computable spam detection features. In *Proceedings of the 4th international workshop on Adversarial information retrieval on the web*, pages 69–76, 2008.
[5] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. Local graph partitioning using pagerank vectors. In *FOCS*, pages 475–486, 2006.
[6] Lars Backstrom and Jure Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 635–644, 2011.
[7] Aleksandar Bojchevski, Johannes Klicpera, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. Scaling graph neural networks with approximate pagerank. In *KDD*, pages 2464–2473, 2020.
[8] Marco Bressan, Enoch Peserico, and Luca Pretto. Sublinear algorithms for local graph centrality estimation. In *FOCS*, pages 709–718, 2018.
[9] Karl Bringmann and Konstantinos Panagiotou. Efficient sampling methods for discrete distributions. In *ICALP*, pages 133–144, 2012.
[10] Ming Chen, Zhewei Wei, Bolin Ding, Yaliang Li, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. Scalable graph neural networks via bidirectional propagation. In *NeurIPS*, 2020.
[11] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *KDD*, pages 257–266, 2019.
[12] Fan Chung. The heat kernel as the pagerank of a graph. *PNAS*, 104(50):19735–19740, 2007.
[13] Fan Chung and Olivia Simpson. Computing heat kernel pagerank and a local clustering algorithm. *European Journal of Combinatorics*, 68:96–119, 2018.
[14] Mustafa Coskun, Ananth Grama, and Mehmet Koyuturk. Efficient processing of network proximity queries via chebyshev acceleration. In *KDD*, pages 1515–1524, 2016.
[15] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005.
[16] Kurt C Foster, Stephen Q Muth, John J Potterat, and Richard B Rothenberg. A faster katz status score algorithm. *Computational & Mathematical Organization Theory*, 7(4):275–285, 2001.
[17] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *WWW*, pages 505–514, 2013.
[18] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NeurIPS*, 2017.
[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.
[20] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. In *NeurIPS*, 2020.
[21] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *Proceedings of the 12th international conference on World Wide Web*, pages 271–279, 2003.
[22] Jinhong Jung, Namyong Park, Sael Lee, and U Kang. Bepi: Fast and memory-efficient method for billion-scale random walk with restart. In *SIGMOD*, pages 789–804, 2017.
[23] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.
[24] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.
[25] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Predict then propagate: Graph neural networks meet personalized pagerank. In *ICLR*, 2019.
[26] Johannes Klicpera, Stefan Weißenberger, and Stephan Günnemann. Diffusion improves graph learning. In *NeurIPS*, pages 13354–13366, 2019.
[27] Kyle Kloster and David F Gleich. Heat kernel based community detection. In *KDD*, pages 1386–1395, 2014.
[28] David Liben-Nowell and Jon M. Kleinberg. The link prediction problem for social networks. In *CIKM*, pages 556–559, 2003.
[29] Dandan Lin, Raymond Chi-Wing Wong, Min Xie, and Victor Junqiu Wei. Index-free approach with theoretical guarantee for efficient random walk with restart query. In *ICDE*, pages 913–924. IEEE, 2020.
[30] Peter Lofgren and Ashish Goel. Personalized pagerank to a target node. *arXiv preprint arXiv:1304.4658*, 2013.
[31] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *KDD*, pages 1105–1114, 2016.
[32] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.
[33] Kijung Shin, Jinhong Jung, Lee Sael, and U. Kang. BEAR: block elimination approach for random walk with restart on large graphs. In *SIGMOD*, pages 1571–1585, 2015.
[34] Daniel A Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *STOC*, pages 81–90, 2004.
[35] Hanzhi Wang, Zhewei Wei, Junhao Gan, Sibo Wang, and Zengfeng Huang. Personalized pagerank to a target node, revisited. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 657–667, 2020.
[36] Sibo Wang, Youze Tang, Xiaokui Xiao, Yin Yang, and Zengxiang Li. Hubppr: Effective indexing for approximate personalized pagerank. *PVLDB*, 10(3):205–216, 2016.
[37] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. FORA: simple and effective approximate single-source personalized pagerank. In *KDD*, pages 505–514, 2017.
[38] Zhewei Wei, Xiaodong He, Xiaokui Xiao, Sibo Wang, Shuo Shang, and Ji-Rong Wen. Topppr: top-k personalized pagerank queries with precision guarantees on large graphs. In *SIGMOD*, pages 441–456. ACM, 2018.
[39] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *ICML*, pages 6861–6871. PMLR, 2019.
[40] Renchi Yang, Xiaokui Xiao, Zhewei Wei, Sourav S Bhowmick, Jun Zhao, and Rong-Hua Li. Efficient estimation of heat kernel pagerank for local clustering. In *SIGMOD*, pages 1339–1356, 2019.
[41] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. In *ICLR*, 2020.
[42] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. Layer-dependent importance sampling for training deep and large graph convolutional networks. In *NeurIPS*, pages 11249–11259, 2019.

Table 5: Hyper-parameters of AGP.

Data set	Learning rate	Dropout	Hidden dimension	Batch size	GDC-AGP	APPNP-AGP	SGC-AGP	AGP		
					t	α	L	a	b	w_i
Yelp	0.01	0.1	2048	$3 \cdot 10^4$	4	0.9	10	-	-	-
Amazon	0.01	0.1	1024	10^5	4	0.2	10	0.8	0.2	$\alpha(1-\alpha)^i, \alpha = 0.2$
Reddit	0.0001	0.3	2048	10^4	3	0.1	10	-	-	-
Papers100M	0.0001	0.3	2048	10^4	4	0.2	10	0.5	0.5	$e^{-t} \cdot \frac{t^i}{i!}, t = 4$

Table 6: URLs of baseline codes.

Methods	URL
GDC	https://github.com/klicperajo/gdc
APPNP	https://github.com/rusty1s/pytorch_geometric
SGC	https://github.com/Tiiiger/SGC
PPRGo	https://github.com/TUM-DAML/pprgo_pytorch
GBP	https://github.com/chennnM/GBP
ClusterGCN	https://github.com/benedekrozemberczki/ClusterGCN

A ADDITIONAL EXPERIMENTAL DESCRIPTIONS

A.1 Local clustering with HKPR

Methods and Parameters. For our method, we set $a = 0, b = 1, w_i = \frac{e^{-t}t^i}{i!}$, and $\boldsymbol{x} = \boldsymbol{e}_s$ in Equation (2) to simulate the HKPR equation $\boldsymbol{\pi} = \sum_{i=0}^{\infty} \frac{e^{-t}t^i}{i!} \cdot \left(\mathbf{AD}^{-1}\right)^i \cdot \boldsymbol{e}_s$. We employ the randomized propagation algorithm 2 with level number $L = O(\log 1/\delta)$ and error parameter $\varepsilon = \frac{2\delta}{L(L+1)}$, where δ is the relative error threshold in Definition 1.1. We use AGP to denote this method. We vary δ from 0.1 to 10^{-8} to obtain a trade-off curve between the approximation quality and the query time. We use the results derived by the Basic Propagation Algorithm 1 with $L = 50$ as the ground truths for evaluating the trade-off curves of the approximate algorithms.

We compare AGP with four local clustering methods: TEA [40] and its optimized version TEA+, ClusterHKPR [13], and the Monte-Carlo method (MC). TEA [40], as the state-of-the-art clustering method, combines a deterministic push process with the Monte-Carlo random walk. Given a graph $G = (V, E)$, and a seed node s, TEA conducts a local search algorithm to explore the graph around s deterministically, and then generates random walks from nodes with residues exceeding a threshold parameter r_{max}. One can manipulate r_{max} to balance the two processes. It is shown in [40] that TEA can achieve $O\left(\frac{t \cdot \log n}{\delta}\right)$ time complexity, where t is the constant heat kernel parameter. ClusterHKPR [13] is a Monte-Carlo based method that simulates adequate random walks from the given seed node and uses the percentage of random walks terminating at node v as the estimation of $\boldsymbol{\pi}(v)$. The length of walks k follows the Poisson distribution $\frac{e^{-t}t^k}{k!}$. The number of random walks need to achieve a relative error of δ in Definition 1.1 is $O\left(\frac{t \cdot \log n}{\delta^3}\right)$. MC [40] is an optimized version of random walk process that sets identical length for each walk as $L = t \cdot \frac{\log 1/\delta}{\log \log 1/\delta}$. If a random walk visit node v at the k-th step, we add $\frac{e^{-t}t^k}{n_r \cdot k!}$ to the propagation results $\hat{\boldsymbol{\pi}}(v)$, where n_r denotes the total number of random walks. The number of random walks to achieve a relative error of δ is also $O\left(\frac{t \cdot \log n}{\delta^3}\right)$. Similar to AGP, for each method, we vary δ from 0.1 to 10^{-8} to obtain a trade-off curve between the approximation quality and the query time. Unless specified otherwise, we set the heat kernel parameter t as 5, following [27, 40]. All local clustering experiments are conducted on a machine with an Intel(R) Xeon(R) Gold 6126@2.60GHz CPU and 500GB memory.

A.2 Node classification with GNN

Datasets. Following [41, 42], we perform inductive node classification on Yelp, Amazon and Reddit, and semi-supervised transductive node classification on Papers100M. More specifically, for inductive node classification tasks, we train the model on a graph with labeled nodes and predict nodes' labels on a testing graph. For semi-supervised transductive node classification tasks, we train the model with a small subset of labeled nodes and predict other nodes' labels in the same graph. We follow the same training/validation/testing split as previous works in GNN [20, 41].

GNN models. We first consider three proximity-based GNN models: APPNP [25],SGC [39], and GDC [26]. We augment the three models with the AGP Algorithm 2 to obtain three variants: APPNP-AGP, SGC-AGP and GDC-AGP. Take SGC-AGP as an example. Recall that SGC uses $\mathbf{Z} = \left(\mathbf{D}^{-\frac{1}{2}}\mathbf{AD}^{-\frac{1}{2}}\right)^L \cdot \mathbf{X}$ to perform feature aggregation, where $\mathbf{X}$ is the $n \times d$ feature matrix. SGC-AGP treats each column of $\mathbf{X}$ as a graph signal $\boldsymbol{x}$ and perform randomized propagation algorithm (Algorithm 2) with predetermined error parameter δ to obtain the the final representation $\mathbf{Z}$. To achieve high parallelism, we perform propagation for multiple columns of $\mathbf{X}$ in parallel. Since APPNP and GDC's original implementation cannot scale on billion-edge graph Papers100M, we implement APPNP and GDC in the AGP framework. In particular, we set $\varepsilon = 0$ in Algorithm 2 to obtain the exact propagation matrix $\mathbf{Z}$, in which case the approximate models APPNP-AGP and GDC-AGP essentially become the exact models APPNP and GDC. We set L=20 for GDC-AGP and APPNP-AGP, and L=10 for SGC-AGP. Note that SGC suffers from the over-smoothing problem when the number of layers L is large [39]. We vary the parameter ε to obtain a trade-off curve between the classification accuracy and the computation time.

Besides, we also compare AGP with three scalable methods: PPRGo [7], GBP [10], and ClusterGCN [11]. Recall that PPRGo is an improvement work of APPNP. It has three main parameters: the number of non-zero PPR values for each training node k, the number of hops L, and the residue threshold r_{max}. We vary the three parameters (k, L, r_{max}) from $(32, 2, 0.1)$ to $(64, 10, 10^{-5})$. GBP decouples the feature propagation and prediction to achieve high scalability. In the propagation process, GBP has two parameters: the propagation threshold r_{max} and the level L. We vary r_{max} from 10^{-4} to 10^{-10}, and set $L = 4$ following [10]. ClusterGCN uses graph sampling method to partition graphs into small parts, and performs the feature propagation on one randomly picked sub-graph in each mini-batch. We vary the partition numbers from 10^4 to 10^5, and the propagation layers from 2 to 4. For AGP, we vary δ from 10^{-5} to 10^{-10}, and tune a, b, w_i for the best performance.

For each method, we apply a neural network with 4 hidden layers, trained with mini-batch SGD. We employ initial residual connection [19] across the hidden layers to facilitate training. We use the trained model to predict each testing node's labels and take the mean accuracy after five runs. For GDC, APPNP, and SGC, we divide the computation time into two parts: the *preprocessing time* for computing $\mathbf{Z}$, and the *training time* for performing mini-batch SGD on $\mathbf{Z}$ until convergence. All the experiments in this section are conducted on a machine with an NVIDIA RTX8000 GPU (48GB memory), Intel Xeon CPU (2.20 GHz) with 40 cores, and 512 GB of RAM. Detailed parameter settings can be founded in the appendix.

Detailed setups. Table 5 summarize the hyper-parameters of AGP. Table 6 summarizes the available URLs of methods we used.