Approximate Graph Propagation Revisited: Dynamic Parameterized Queries, Tighter Bounds and Dynamic Updates

Zhuowei Zhao
The University of Melbourne
zhuoweiz1@student.unimelb.edu.au

Zhuo Zhang
The University of Melbourne
zhuo.zhang@student.unimelb.edu.au

Hanzhi Wang
University of Copenhagen
hanzhi.hzwang@gmail.com

Junhao Gan
The University of Melbourne
junhao.gan@unimelb.edu.au

Zhifeng Bao
The University of Queensland
zhifeng.bao@uq.edu.au

Jianzhong Qi
The University of Melbourne
jianzhong.qi@unimelb.edu.au

ABSTRACT

We revisit *Approximate Graph Propagation* (AGP), a unified framework that captures various graph propagation tasks, such as PageRank, Personalized PageRank, feature propagation in Graph Neural Networks, and graph-based Retrieval-Augmented Generation. Our work focuses on the settings of *dynamic graphs* and *dynamic parameterized queries*, where the underlying graphs evolve over time (updated by edge insertions or deletions) and the input query parameters are specified on the fly to fit application needs. Our first contribution is an interesting observation that the SOTA solution, *AGP-Static*, can be adapted to support dynamic parameterized queries; however, several challenges remain unresolved. Firstly, the query time complexity of AGP-Static is based on an assumption of using an optimal algorithm for its subset sampling. Unfortunately, back to that time, such an algorithm did not exist; without such an optimal algorithm, an extra $O(\log^2 n)$ factor is required in the query complexity, where n is the number of vertices in the graphs. Secondly, AGP-Static performs poorly on dynamic graphs, taking $O(n \log n)$ time to process each update. To address these challenges, we propose a new algorithm, *AGP-Static++*, which is simpler yet reduces roughly a factor of $O(\log^2 n)$ in the query complexity while preserving the approximation guarantees of AGP-Static. However, AGP-Static++ still requires $O(n)$ time per update. To better support dynamic graphs, we further propose *AGP-Dynamic*, which achieves $O(1)$ *amortized time per update*, significantly improving the aforementioned $O(n)$ per-update bound, while still preserving the query complexity and approximation guarantees. Last, our comprehensive experiments validate the theoretical improvements: compared to the baselines, our algorithm achieves speedups of up to 177× on update and 10× on query efficiency.

PVLDB Reference Format:
Zhuowei Zhao, Zhuo Zhang, Hanzhi Wang, Junhao Gan, Zhifeng Bao, and Jianzhong Qi. Approximate Graph Propagation Revisited: Dynamic Parameterized Queries, Tighter Bounds and Dynamic Updates. PVLDB, 19(1): XXX-XXX, 2026.
doi:XX.XX/XXX.XX

Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/alvinzhaowei/AGP-dynamic.

1 INTRODUCTION

Node proximity evaluations, such as PageRank [5, 18] and Personalized PageRank [27], have shown great usefulness in various graph mining and machine learning tasks, including but not limited to link prediction [27], graph representation learning [34], spam detection [16], and web search [27]. Recent studies have integrated node proximity with Graph Neural Networks (GNNs) to enhance scalability, simplifying the original GNN architecture by replacing feature transformation across multiple layers with a feature propagation step [34]. Depending on concrete applications, different node proximity models are deployed. For example, Personalized PageRank [27] is used in PPRGo [2], while heat kernel PageRank [9] is adopted in *graph diffusion convolution* (GDC) [13]. Recently, Wang *et al.* [32] unify a wide range of node proximity models into the following *graph propagation* equation:

$$\boldsymbol{\pi} = \sum_{i=0}^{\infty} w_i \cdot (\mathbf{D}^{-\mathbf{a}}\mathbf{A}\mathbf{D}^{-\mathbf{b}})^i \cdot \mathbf{x}, \tag{1}$$

where $\boldsymbol{\pi}$ is the node proximity evaluation result vector, $\mathbf{A}$ and $\mathbf{D}$ are the adjacency matrix and the diagonal degree matrix of the underlying undirected graph $G = \langle V, E \rangle$, respectively, while a, b, w_i and $\mathbf{x}$ are query parameters specifying the node proximity model. As shown in Table 1, with different parameter settings, Equation (1) captures a range of graph propagation applications.

In most cases, computing the exact solution to Equation (1) can be expensive. To achieve better efficiency, Wang *et al.* [32] propose the notion of (δ, c)-approximation, which only aims to provide approximations within a c-relative error to entries where $\boldsymbol{\pi}(v) \geq \delta$, while ignoring the errors for entries with small values, i.e., $\boldsymbol{\pi}(v) < \delta$. They further proposed a state-of-the-art algorithm, called *Approximate Graph Propagation* (denoted as *AGP-Static*), for computing (δ, c)-approximate results for queries on *static graphs with fixed and pre-specified parameters w_i's, a and b*.

SOTA's Limitation 1: the Static Setting. Unfortunately, such *static* setting of *AGP-Static* with static graphs and pre-specified query parameters means that *AGP-Static* can only work on a fixed

Table 1: Graph Propagation Applications

Algorithm	a	b	w_i	$\mathbf{x}$
L-hop transition probability	0	1	$w_i = 0 (i \neq L), W_L = 1$	one-hot vector
PageRank [27]	0	1	$\alpha(1-\alpha)^i$	$\left[\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right]^T$
Personalized PageRank [27]	0	1	$\alpha(1-\alpha)^i$	one-hot vector
Single-target PPR [26]	1	0	$\alpha(1-\alpha)^i$	one-hot vector
Heat kernel PageRank [5]	0	1	$e^{-t} \cdot \frac{t^i}{i!}$	one-hot vector
Simplifying graph convolutional networks [34]	$\frac{1}{2}$	$\frac{1}{2}$	$w_i = 0 (i \neq L), W_L = 1$	the graph signal
Approx-personalized propagation [12]	$\frac{1}{2}$	$\frac{1}{2}$	$\alpha(1-\alpha)^i$	the graph signal
Graph diffusion convolution [13]	$\frac{1}{2}$	$\frac{1}{2}$	$e^{-t} \cdot \frac{t^i}{i!}$	the graph signal

and pre-specified node proximity model on a static graph, which indeed has significantly limited its applicability in real-world applications, especially, when the underlying graph G is frequently updated and the support of varying node proximity evaluations is required, as illustrated in the case study below.

Case Study: A Fully Dynamic Scenario. In recent years, Large Language Models (LLMs) such as DeepSeek [25], LLaMA [14], and GPT [1] have achieved great success in various applications and have become a trending topic in AI research [4, 20, 24, 30]. Along the process, Retrieval-Augmented Generation (RAG), a technique to enhance the trustworthiness of LLMs through the integration of external knowledge [7, 23, 29], has attracted increasing interest. A notable trend is the use of graph-based RAG [6, 15, 17, 33, 35], which leverages graph-structured data, namely, *knowledge graphs*, as an external knowledge source and has shown promising results. Specifically, a knowledge graph not only stores individual entities but also captures the relationships between them through edges. When a user asks a question to an LLM, graph-based RAG first maps the question keywords to vertices in the knowledge graph. Starting from these vertices as sources, the RAG process retrieves vertices that have high relevance to these sources.

For example, when a question "What causes Alzheimer's disease?" is submitted, the AI system will identify one or more vertices in the knowledge graph that best match the query keywords (e.g., the vertices related to "Alzheimer's disease"). It then attempts to retrieve the most relevant vertex candidate set from the knowledge graph, measured under a certain node proximity model, e.g., Personalized PageRank (PPR), which computes the probability $\pi[v]$ of a random walk from a given source vertex s to each vertex v in the graph. A larger $\pi[v]$ value indicates stronger relevance of v to s. The PPR values can thus be used to select the top-k most relevant vertices (e.g., "beta-amyloid plaques", "tau protein" or "neuro degeneration" for the Alzheimer's disease example). The retrieved vertices (and possibly their attributes or neighbors) are then compiled into a context document and fed into an LLM to generate an answer based on both the question and the graph-derived context.

This kind of graph-based RAG introduces new challenges for graph propagation. First, the knowledge graphs can be updated frequently requiring efficient update support. Second, different questions may require different types of node proximity evaluations. For example, when a user asks, *"Who are the people most affected by Donald Trump?"*, a single-source PPR can be used to model Trump's influence. In contrast, for a question like *"Who are the people that most affect Trump?"*, a single-target PPR is needed. For question *"What is the importance of Trump?"*, it can be modeled as computing the PageRank of Trump. Therefore, an ideal graph propagation algorithm should be able to support queries (with respect to Equation (1)) with parameters w_i's, a and b given on the fly, such that the node proximity model can be specified in queries.

SOTA's Limitation 2: Looseness on Theoretical Bounds. Besides the static setting, *AGP-Static* also suffers from several limitations in their theoretical analysis. First, in [32], the running time analysis of *AGP-Static* is based on an assumption that an optimal algorithm for solving a *special parameterized subset sampling problem* is adopted. However, such an algorithm did not exist back to the time when [32] was published until a very recent work [10]. Indeed, the actual implementation of *AGP-Static* adopts a compromising subset sampling algorithm which leads to a blow-up factor of $O(\log^2 n)$ in the claimed query time complexity, where n is the number of vertices in the graph. Here, a $O(\log n)$ factor in this blow up comes from the usage of power-of-two bucketing technique, while the other $O(\log n)$ factor is introduced by the need of generating binomial random variates in their subset sampling algorithm, where according to the best known result [8], generating each binomial random variate takes $O(\log n)$ expected time[1].

To address the above challenges and limitations, we made the following technical contributions:

- **A Strengthened Version of *AGP-Static*.** We observe that *AGP-Static* indeed can support dynamic queries with parameters given on the fly. We strengthen the algorithm to support each graph update (either an edge insertion or deletion) in $O(d_{\max} \log n) = O(n \log n)$ time, where $d_{\max}$ and n are the maximum degree and the number of vertices in the current graph G. We further show a precise expected query time complexity of *AGP-Static*, $O(\log^2 n \cdot \frac{L^2}{\delta} \cdot Z + n)$, where L is the number of iterations which is often equal to $O(\log n)$, and Z is an output-size sensitive term.
- **Our Solutions for Enhanced Efficiency.** We propose *AGP-Static++* which adopts a simpler subset sampling algorithm yet is more efficient in practice. With a careful analysis, we prove that the expected query time complexity of *AGP-Static++* is bounded by $O(\log n \cdot \frac{L}{\delta} \cdot Z + n)$, a factor of $O(L \log n) \approx O(\log^2 n)$ improvement over *AGP-Static* since the first term often dominates the second in the complexity. Moreover, *AGP-Static++* also improves over *AGP-Static* in terms of update efficiency, though it still takes $O(d_{\max}) = O(n)$ time to process each graph update.
- **Our Fully-Dynamic Algorithm.** To support efficient graph updates, we propose our ultimate algorithm, *AGP-Dynamic*, which not only supports each update in $O(1)$ *amortized* time, but also achieves the same expected query complexity and preserves the same approximation guarantees as *AGP-Static++*.
- **Extensive Experimental Study.** We conduct experiments on nine real-world datasets. The experiment results show that our *AGP-Dynamic* outperforms the SOTA *AGP-Static* by up to 117 times on update efficiency with different update patterns and insertion-deletion ratios, and by up to 10 times on query efficiency with small variance on arbitrary input parameters.

[1]Whether a binomial random variate can be generated in $O(1)$ expected time still remains an open problem [11, 19, 21].

2 PRELIMINARIES

2.1 Problem Definition

Consider an undirected and unweighted graph $G = \langle V, E \rangle$, where V is a set of n vertices and E is a set of m edges. The *neighborhood* of a vertex u, denoted as $N[u] = \{v \in V \mid (u, v) \in E\}$, is the set of all vertices adjacent to u. The *degree* of u, denoted by $d_u = |N[u]|$, represents the number of neighbors of u. The adjacency matrix and diagonal degree matrix of G are denoted as $\mathbf{A}$ and $\mathbf{D}$. We aim to solve the following **Unified Graph Propagation Equation** (Equation (1)):

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

where the query parameters a, b, w_i and $\mathbf{x}$ satisfy the following conditions:

- $a, b \in [0, 1]$ are *constants* such that $a + b \geq 1$;
- the weight parameters w_i's satisfy:
  - $w_i \geq 0$ for all i, and $\sum_{i=0}^{\infty} w_i = 1$ (this can be achieved by normalizing $\sum_{i=0}^{\infty} w_i$ to 1);
  - $w_i \leq \lambda^i$ for all $i \geq L_0$, where $L_0 \geq 0$ and $\lambda \geq 1$ are some constants;
  - w_i's are input as an *oracle* O_w with which these information can be returned in $O(1)$ time: (i) the value of w_i for all integer $i \geq 0$, and (ii) the value of $L_{O_w}(\Delta)$ which is the smallest integer k such that $\sum_{i=k}^{\infty} w_i \leq \Delta$ for the given $0 \leq \Delta \leq 1$.
- $\mathbf{x}$ is an n-dimensional vector such that $\|\mathbf{x}\|_1 = 1$ and $\mathbf{x}_i \geq 0, \forall\, i = 1, 2, 3, \cdots, n$.

It can be verified that all the node proximity models in Table 1 satisfy the above conditions, optionally after normalization. In what follows, the node proximity model parameterized by a, b, O_w and $\mathbf{x}$ is called a *query* and denoted by $q(a, b, O_w, \mathbf{x})$.

(δ, c)-Approximation. Given a threshold $\delta > 0$ and a constant $c > 0$, an estimation of $\boldsymbol{\pi}$, denoted by $\hat{\boldsymbol{\pi}}$, is an (δ, c)-approximation of $\boldsymbol{\pi}$ if it satisfies:

$$\text{for all } v \in V \text{ with } |\boldsymbol{\pi}(v)| > \delta, \quad |\boldsymbol{\pi}(v) - \hat{\boldsymbol{\pi}}(v)| \leq c \cdot \boldsymbol{\pi}(v)\,.$$

We have the following fact:

Fact 1. *For every query $q(a, b, O_w, \mathbf{x})$, $\hat{\pi} = \sum_{i=0}^{L} w_i \cdot (\mathbf{D}^{-\mathbf{a}}\mathbf{A}\mathbf{D}^{-\mathbf{b}})^i \cdot \mathbf{x}$ is an (δ, c)-approximation of $\boldsymbol{\pi}$, where $L = L_{O_w}(c \cdot \delta) \in O(\log_{\lambda} \frac{1}{c \cdot \delta}) = O(\log \frac{1}{\delta})$.*

Proof. By the fact that $\mathbf{A}\mathbf{D}^{-1}$ has an eigenvalue of 1, we have: $\sum_{i=L+1}^{\infty} w_i \cdot (\mathbf{D}^{-\mathbf{a}}\mathbf{A}\mathbf{D}^{-\mathbf{b}})^i \cdot \mathbf{x} = \sum_{i=L+1}^{\infty} w_i \cdot (\mathbf{A}\mathbf{D}^{-(a+b)})^i \cdot \mathbf{x} \leq \sum_{i=L+1}^{\infty} w_i \cdot (\mathbf{A}\mathbf{D}^{-1})^i \cdot \mathbf{x} \leq \sum_{i=L+1}^{\infty} w_i \cdot \mathbf{x} \leq \sum_{i=L+1}^{\infty} w_i \cdot \|\mathbf{x}\|_1 \leq c \cdot \delta$. □

Problem Definition. Given an undirected graph $G = \langle V, E \rangle$, the problem of *Dynamic Approximate Graph Propagation* (DAGP) is to design an algorithm which:

- for every query $q(a, b, O_w, \mathbf{x})$, approximation parameters $\delta > 0$ and $c > 0$, with at least a constant probability, returns an (δ, c)-approximation $\hat{\boldsymbol{\pi}}$, and
- supports each graph update (i.e., either an edge insertion or deletion) efficiently.

2.2 The State-Of-The-Art Algorithm

According to Fact 1, one can compute an (δ, c)-approximation $\hat{\boldsymbol{\pi}}$ by running the Power Method [31] for L iterations, which takes $O(m \cdot L)$ time, where m is the number of edges in G. To improve the query efficiency, Wang *et al.* [32] proposed the SOTA algorithm, *AGP-Static*, which simply pushes the "probability mass" from some vertices u to a random subset of their neighbors in each iteration. The pseudocode of *AGP-Static* is in Algorithm 1.

Algorithm 1: AGP-Static

Input: $q(a, b, O_w, \mathbf{x})$, parameters δ and c for (δ, c)-approximation
Output: $\hat{\boldsymbol{\pi}}$, an (δ, c)-approximation of $\boldsymbol{\pi}$

1 $L \leftarrow L_{O_w}(c \cdot \delta)$; $\varepsilon \leftarrow \frac{c^2\delta}{2L^2}$;
2 $\hat{\mathbf{r}}^{(0)} \leftarrow \mathbf{x}$;
3 **for** $i = 0$ *to* $L - 1$ **do**
4 **for** *each* $u \in V$ *with* $\hat{\mathbf{r}}^{(i)}(u) \neq 0$ **do**
5 Scan the sorted neighborhood
6 **for** *each* $v \in N[u]$ *and* $d_v \leq \left(\frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\mathbf{r}}^{(\mathbf{i})}(\mathbf{u})}{d_u^b}\right)^{\frac{1}{a}}$ **do**
7 $\hat{\mathbf{r}}^{(i+1)}(v) \leftarrow \hat{\mathbf{r}}^{(i+1)}(v) + \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\mathbf{r}}^{(i)}(u)}{d_v^a \cdot d_u^b}$;
8 Sample each remaining neighbor $v \in N[u]$ with probability $p_{u,v} = \frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\mathbf{r}}^{(\mathbf{i})}(\mathbf{u})}{d_v^a d_u^b}$
9 **for** *each* v *sampled* **do**
10 $\hat{\mathbf{r}}^{(i+1)}(v) \leftarrow \hat{\mathbf{r}}^{(i+1)}(v) + \varepsilon$;
11 $\hat{\mathbf{q}}^{(i)}(u) \leftarrow \hat{\mathbf{q}}^{(i)}(u) + \frac{w_i}{Y_i} \cdot \hat{\mathbf{r}}^{(i)}(u)$;
12 $\hat{\boldsymbol{\pi}} \leftarrow \hat{\boldsymbol{\pi}} + \hat{\mathbf{q}}^{(i)}$;
13 $\hat{\mathbf{q}}^{(L)} = \frac{w_L}{Y_L} \cdot \hat{\mathbf{r}}^{(L)}$; and $\hat{\boldsymbol{\pi}} \leftarrow \hat{\boldsymbol{\pi}} + \hat{\mathbf{q}}^{(L)}$;
14 **return** $\hat{\boldsymbol{\pi}}$

Given a query $q(a, b, O_w, \mathbf{x})$, *AGP-Static* introduces the notions of *residue* and *reserve vector* at the i^{th} iteration (where $i = 0, 1, \ldots, L$) to facilitate iterative computation:

- *residue* $\mathbf{r}^{(\mathbf{i})} = Y_i \cdot (\mathbf{D}^{-\mathbf{a}}\mathbf{A}\mathbf{D}^{-\mathbf{b}})^i \cdot \mathbf{x}$, where $Y_i = \sum_{k=i}^{\infty} w_k$;
- *reserve vector* $\mathbf{q}^{(\mathbf{i})} = \frac{w_i}{Y_i} \cdot \mathbf{r}^{(\mathbf{i})} = w_i \cdot (\mathbf{D}^{-\mathbf{a}}\mathbf{A}\mathbf{D}^{-b})^i \cdot \mathbf{x}$.

Moreover, the residue $\mathbf{r}^{(i+1)}$ in the next step (i.e., iteration) can be computed from the previous step by:

$$\mathbf{r}^{(i+1)} = \frac{Y_{i+1}}{Y_i} \cdot (\mathbf{D}^{-\mathbf{a}}\mathbf{A}\mathbf{D}^{-b}) \cdot \mathbf{r}^{(i)},$$

and so does reserve vector $\mathbf{q}^{(i)}$. An (δ, c)-approximation $\hat{\boldsymbol{\pi}}$ can thus be obtained by $\hat{\boldsymbol{\pi}} = \sum_{i=0}^{L} \mathbf{q}^{(i)} = \sum_{i=0}^{L} w_i \cdot (\mathbf{D}^{-\mathbf{a}}\mathbf{A}\mathbf{D}^{-\mathbf{b}})^i \cdot \mathbf{x}$.

Before answering any query, *AGP-Static* needs to construct two data structures on the neighborhood $N[u]$ for each vertex $u \in V$: (i) a sorted list of $N[u]$ by their degrees, and (ii) a specific data structure (which is introduced below) to support efficient *subset sampling* from $N[u]$. To answer a query, *AGP-Static* performs L iterations; in the i^{th} iteration, for each vertex v in the neighborhood $N[u]$:

- by scanning the sorted neighbor list: if d_v is small (Line 5) , propagate the exact amount of probability mass with respect to $\hat{\mathbf{r}}^{(i)}(u)$ to $\hat{\mathbf{r}}^{(i+1)}(v)$ (Line 6);
- by the subset sampling structure: sample each remaining neighbor v with a certain probability (Line 7) , and if v is sampled, a

probability mass of $\varepsilon = O(\frac{\delta}{L^2})$ is propagated to $\hat{\mathbf{r}}^{(i+1)}(v)$ (Lines 9 - 10).

The Subset Sampling Data Structure in *AGP-Static*. To sample the neighbors efficiently, for each $u \in V$, *AGP-Static* constructs a data structure $B[u]$, which consists of $\lfloor \log_2 n \rfloor + 1$ *power-of-two* buckets. Each bucket is denoted as $B[u]_j$, where j is the bucket index. Each neighbor $v \in N[u]$ is assigned to the bucket $B[u]_j$ such that $2^j \le d_v < 2^{j+1}$. The structure $B[u]$ is implemented as a sorted list of all the non-empty buckets and the vertices (i.e., u's neighbors) therein. Therefore, it takes $O(d_u)$ space for each vertex and $O(n+m)$ for the entire graph to store this structure.

To decide whether a neigbhor $v \in N[u]$ is sampled, *AGP-Static* performs a subset sampling on $N[u]$. For each bucket $B[u]_j \in B[u]$,

- generate a *binomial* random variate $X_j \sim \text{Bin}(|B[u]_j|, p^*)$, where $p^* = \frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\mathbf{r}}^{(i)}(u)}{(2^j)^a d_u^b}$, and X_j is the target number of distinct vertices in $B[u]_j$ to be sampled with probability p^*;
- uniformly at random sample X_j *distinct* vertices in $B[u]_j$;
- each sampled neighbor v is accepted with probability $\frac{p_{u,v}}{p^*}$, where $p_{u,v} = \frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\mathbf{r}}^{(i)}(u)}{d_v^a d_u^b}$.

Analysis. According to the SOTA result [8], generating a binomial random variate with probability p^* takes $O(\log n)$ expected time. As there can be $O(\log n)$ buckets in $B[u]$, and by the fact that each neighbor taken from the bucket will be accepted by probability at least $\frac{p_{u,v}}{p^*} = (\frac{2^j}{d_v})^a \ge \frac{1}{2^a} \in \Omega(1)$, the neighbor subset sampling for each vertex $u \in V$ takes $O(\log^2 n + \mu_u)$ expected time, where μ_u is the expected size of the sample set.

Observation 1. *While* AGP-Static *was originally proposed [32] for queries with parameters a and b given in advance and fixed, it can support queries with a and b given on the fly.*

3 OUR *AGP-STATIC++* ALGORITHM

In a nutshell, our *AGP-Static++* follows the algorithm framework of *AGP-Static*. We also group the neighbors of each vertex u into a sorted list of power-of-two buckets, $B[u]$, by their degrees. However, there are also substantial differences that are the key to our practical and theoretical improvements:

- *Eliminating the Sorted Lists:* We propose a new subset sampling algorithm that removes the need of a sorted list in each neighborhood.
- *Improved Parameter Setting:* In *AGP-Static++*, the parameter ε is set to $O(\frac{\delta}{L})$, reducing a factor of $O(L)$ from that in *AGP-Static*. With our tighter analysis (Section 4.2), we show that *AGP-Static++* achieves the same approximation guarantees as *AGP-Static* does. This refinement improves the query time complexity by a factor of $O(L)$ compared to *AGP-Static*.
- *Optimized Subset Sampling Technique:* We employ a different subset sampling method that achieves a complexity of $O(\log n + \mu)$, where μ denotes the expected output size. This removes a $O(\log n)$ factor from the query complexity of *AGP-Static*.

3.1 The Subset Sampling Algo's in *AGP-Static++*

AGP-Static++ consolidates the propagation procedures within the neighborhood of vertex u into a single subset sampling process (Lines 3–6). This eliminates the need of the sorted list, which not only simplifies the algorithm in the static case but is also essential for handling graph updates efficiently, as maintaining the sorted list can be costly (e.g., $O(\log n)$ per update). Our proposed subset sampling method operates in $O(\mu + \log n)$ expected time, where μ is the expected output size. Specifically, it works as follows.

Algorithm 2: *AGP-Static++*

Input: $q(a, b, O_w, \mathbf{x})$, parameters δ and c for (δ, c)-approximation
Output: $\hat{\boldsymbol{\pi}}$, an (δ, c)-approximation of $\boldsymbol{\pi}$

1 $L \leftarrow L_{O_w}(c \cdot \delta)$; $\varepsilon \leftarrow \frac{c^2\delta}{2(L+1)}$; $\hat{\mathbf{r}}^{(0)} \leftarrow \mathbf{x}$;
2 **for** $i = 0$ *to* $L-1$ **do**
3 **for** *each* $u \in V$ *with* $\hat{\mathbf{r}}^{(i)}(u) \neq 0$ **do**
4 obtain a subset T sampled from neighbors in $N[u]$ such that each neighbor is sampled independently with probability $p_{u,v} = \min\left\{1, \frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\mathbf{r}}^{(i)}(u)}{d_v^a d_u^b}\right\}$;
5 **for** *each* $v \in T$ **do**
6 $\hat{\mathbf{r}}^{(i+1)}(v) \leftarrow \hat{\mathbf{r}}^{(i+1)}(v) + \max\left\{\varepsilon, \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\mathbf{r}}^{(i)}(u)}{d_v^a d_u^b}\right\}$
7 $\hat{\mathbf{q}}^{(i)}(u) \leftarrow \hat{\mathbf{q}}^{(i)}(u) + \frac{w_i}{Y_i} \cdot \hat{\mathbf{r}}^{(i)}(u)$;
8 $\hat{\boldsymbol{\pi}} \leftarrow \hat{\boldsymbol{\pi}} + \hat{\mathbf{q}}^{(i)}$;
9 $\hat{\mathbf{q}}^{(L)} = \frac{w_L}{Y_L} \cdot \hat{\mathbf{r}}^{(L)}$; and $\hat{\boldsymbol{\pi}} \leftarrow \hat{\boldsymbol{\pi}} + \hat{\mathbf{q}}^{(L)}$;
10 **return** $\hat{\boldsymbol{\pi}}$

In the i^{th} iteration, for each neighborhood $N[u]$ with $\hat{\mathbf{r}}^{(i)}(u) \neq 0$:

- Compute a shifting factor $s_u^{(i)} = \frac{1}{\varepsilon} \cdot \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\mathbf{r}}^{(i)}(u)}{d_u^b}$.
- For each bucket $B[u]_j$ in $B[u]$, let $p^* = \min\left\{1, \frac{s_u^{(i)}}{(2^j)^a}\right\}$. If $p^* = 1$, all the vertices in $B[u]_j$ are taken in T; otherwise, sample each neighbor independently with probability p^*:
  - initialize $y \leftarrow 0$;
  - generate j from the bounded geometric distribution $Geo(p^*, |B[u]_j|)$, and set $y \leftarrow y + j$;
  - if $y > |B[u]_j|$, stop; otherwise, take the y^{th} vertex in $B[u]_j$ and repeat by generating another j from the second step.
- For each $v \in T$, accept v with probability $p_{\text{ac}} = \frac{p_{u,v}}{p^*}$ and propagate a probability mass of $\max\left\{\varepsilon, \frac{Y_{i+1}}{Y_i} \cdot \frac{\hat{\mathbf{r}}^{(i)}(u)}{d_v^a d_u^b}\right\}$ to $\mathbf{r}^{(i+1)}(v)$.

The correctness of the above subset sampling method with geometric distribution comes from the fact that $j \sim Geo(p^*, |B[u]_j|)$ follows the same distribution of the first index of success in a Bernoulli trial with success probability p^*, which, in turn, is equivalent to independently flipping a coin of with success probability of p^* for each neighbor in the bucket.

Analysis. It is known that a random variate from a bounded geometric distribution $j \sim Geo(p^*, N)$ can be generated in $O(1)$ expected time in the Word RAM model [3]. The probability p^* used to sample in each bucket is at most $2^a \in O(1)$ times the required probability $p_{u,v}$ – recall that $p^* = s_u \cdot \frac{1}{(2^j)^a}$, $p_{u,v} = s_u \cdot \frac{1}{d_v^a}$ and $2^j \le d_v < 2 \cdot 2^j$. Given any vertex u, there are at most $O(\log n)$ buckets in $B[u]$,

hence the total time complexity for the subset sampling within each neighborhood is bounded by $\sum_k O(\mu_k + 1) = O(\mu + \log n)$, where μ_k is the expected output size for the k^{th} bucket in $B[u]$, and μ is the expected sample size from $N[u]$.

4 THEORETICAL ANALYSIS

4.1 Query Running Time Analysis

We now analyze the complexity of *AGP-Static* and *AGP-Static++*. Without loss of generality, we denote the subset sampling cost within the neighborhood of a vertex u by $O(\mu_u + c_{\text{so}})$, where μ_u is the expected sample size and c_{so} is the *non-chargeable* overhead. In particular, $c_{\text{so}} = O(\log^2 n)$ for *AGP-Static*, while $c_{\text{so}} = O(\log n)$ for our *AGP-Static++*.

Theorem 1. *The expected query time complexity of AGP algorithms (e.g.,* AGP-Static *and* AGP-Static++*) is bounded by*

$$O\left(c_{so} \cdot \frac{1}{\varepsilon} \sum_{i=1}^{L} \|Y_i(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b})^i \cdot \mathbf{x}\|_1 + E[C_{init}]\right),$$

where $E[C_{init}]$ is the expected cost of initialization, i.e., reading the query input $q(a, b, O_w, \mathbf{x})$.

Proof. Besides the initialization cost (C_{init}), the query running time mainly consists of these two costs:

- the propagation cost on each edge at each iteration, denoted as $c^{(i)}(u, v)$, meaning the cost to propagate the probability mass from u to v at the i^{th} iteration;
- the subset sampling non-chargeable overhead cost at each neighborhood at each iteration, denoted as $c_{\text{so}}^{(i)}(u)$, meaning the non-chargeable subset sampling overhead cost for $N[u]$ at the i^{th} iteration. Here, we only need to consider the non-chargeable overhead as the output cost and the chargeable overhead can be charged to $c^{(i)}(u, v)$ for those neighbors v sampled in $N[u]$ at step i.

We first analyze the expected cost of $c^{(i)}(u, v)$. According to Line 4 in Algorithm 2, each neighbor is sampled independently by probability $\min\{1, p_{u,v}\}$, and each propagation cost is $O(1)$. Thus, the expected cost of $c^{(i)}(u, v)$ conditioned on $\hat{\mathbf{r}}^{(i-1)}(u)$, i.e., $\mathrm{E}[c^{(i)}(u, v)|\hat{\mathbf{r}}^{(i-1)}(u)] = \min\{1, p_{u,v}\} \le p_{u,v}$, where $p_{u,v} = \frac{1}{\varepsilon} \cdot \frac{Y_i}{Y_{i-1}} \cdot \frac{\hat{\mathbf{r}}^{(i-1)}(u)}{d_v^a d_u^b}$.

It is known that the following fact holds:

Fact 2 ([32]). *$\hat{\mathbf{r}}$ is an unbiased estimation of $\mathbf{r}$, i.e., $E[\hat{\mathbf{r}}^{(i-1)}(u)] = \mathbf{r}^{(i-1)}(u)$.*

By Fact 2, we then have:

$$\begin{aligned}
\mathrm{E}[c^{(i)}(u, v)] &= \mathrm{E}[\mathrm{E}[c^{(i)}(u, v)|\hat{\mathbf{r}}^{(i-1)}(u)]] \\
&= \sum_j \Pr[\hat{\mathbf{r}}^{(i-1)}(u)_j] \cdot \mathrm{E}[c^{(i)}(u, v)|\hat{\mathbf{r}}^{(i-1)}(u)_j] \\
&\le \sum_j \Pr[\hat{\mathbf{r}}^{(i-1)}(u)_j] \cdot \frac{1}{\varepsilon} \cdot \frac{Y_i}{Y_{i-1}} \cdot \frac{\hat{\mathbf{r}}^{(i-1)}(u)_j}{d_v^a d_u^b} \\
&= \frac{1}{\varepsilon} \cdot \frac{Y_i}{Y_{i-1}} \cdot \frac{\sum_j \Pr[\hat{\mathbf{r}}^{(i-1)}(u)_j] \cdot \hat{\mathbf{r}}^{(i-1)}(u)_j}{d_v^a d_u^b} \\
&\le \frac{1}{\varepsilon} \cdot \frac{Y_i}{Y_{i-1}} \cdot \frac{\mathrm{E}[\hat{\mathbf{r}}^{(i-1)}(u)]}{d_v^a d_u^b} = \frac{1}{\varepsilon} \cdot \frac{Y_i}{Y_{i-1}} \cdot \frac{\mathbf{r}^{(i-1)}(u)}{d_v^a d_u^b}.
\end{aligned}$$

Next, we analyze $\mathrm{E}[c_{\text{so}}^{(i)}(u)]$. To facilitate our analysis, we define $I^{(i)}(u)$ as the indicator of whether $\hat{\mathbf{r}}^{(i)}(u) > 0$ is true or not. Specifically, $I^{(i)}(u) = 1$ when $\hat{\mathbf{r}}^{(i)}(u) > 0$, and $I^{(i)}(u) = 0$ otherwise. When $\hat{\mathbf{r}}^{(i)}(u) > 0$, only if at the $(i-1)^{\text{th}}$ step, at least one of u's neighbors has propagated to u. In other words, u must have been sampled from the neighborhood $N[v]$ of at least one of its neighbors $v \in N[u]$. Since u is sampled independently from each of its neighbors v with probability $\min\{1, p_{v,u}\}$, by Union Bound, when $i \ge 1$, $\Pr[I^{(i)}(u) = 1] \le \sum_{v \in N[u]} \frac{1}{\varepsilon} \cdot \frac{Y_i}{Y_{i-1}} \cdot \frac{\hat{\mathbf{r}}^{(i-1)}(v)}{d_u^a d_v^b}$. Therefore, we have $\mathrm{E}\left[I^{(i)}(u)\right] = \mathrm{E}\left[\mathrm{E}\left[I^{(i)}(u)|\hat{\mathbf{r}}^{(i-1)}\right]\right] \le \sum_{v \in N[u]} \frac{1}{\varepsilon} \cdot \frac{Y_i}{Y_{i-1}} \cdot \frac{\mathrm{E}[\hat{\mathbf{r}}^{(i-1)}(v)]}{d_u^a d_v^b} = \frac{1}{\varepsilon} \cdot \mathbf{r}^{(i)}(u)$; recall that $\hat{\mathbf{r}}$ is an unbiased estimation.

Putting these together, the expected overall cost can be computed as follows:

$$\begin{aligned}
\mathrm{E}_{\text{cost}} &= \sum_{i=1}^{L} \sum_{v \in V} \sum_{u \in N[v]} \mathrm{E}[c^{(i)}(u, v)] + \sum_{i=1}^{L-1} \sum_{v \in V} \mathrm{E}[c_{so}^{(i)}(v)] + \mathrm{E}[C_{\text{init}}] \\
&\le \sum_{i=1}^{L} \sum_{v \in V} \sum_{u \in N[v]} \frac{1}{\varepsilon} \cdot \frac{Y_i}{Y_{i-1}} \cdot \frac{\mathbf{r}^{(i-1)}(u)}{d_v^a d_u^b} \\
&\quad + \sum_{i=1}^{L-1} \sum_{v \in V} \mathrm{E}[I^{(i)}(v)] \cdot c_{\text{so}} + \mathrm{E}[C_{\text{init}}] \\
&\le \frac{1}{\varepsilon} \sum_{v \in V} \sum_{i=1}^{L} \mathbf{r}^{(i)}(v) + c_{\text{so}} \cdot \frac{1}{\varepsilon} \sum_{v \in V} \sum_{i=1}^{L-1} \mathbf{r}^{(i)}(v) + \mathrm{E}[C_{\text{init}}] \\
&= O\left(c_{\text{so}} \cdot \frac{1}{\varepsilon} \sum_{i=1}^{L} \|Y_i(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b})^i \cdot \mathbf{x}\|_1 + \mathrm{E}[C_{\text{init}}]\right),
\end{aligned}$$

where the last bound follows the definition of $\mathbf{r}^{(i)}(u)$. □

By substituting the values of c_{so} and ε of each algorithm into Theorem 1, the concrete expected query time complexity follows.

Corollary 1. *The expected query time complexity of* AGP-Static *is bounded by*

$$O\left(\log^2 n \cdot \frac{L^2}{\delta} \sum_{i=1}^{L} \|Y_i(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b})^i \cdot \mathbf{x}\|_1 + E[C_{init}]\right).$$

Corollary 2. *The expected query time complexity of* AGP-Static++ *is bounded by*

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

Incorporating DPSS. The subset sampling cost in *AGP-Static++* can be further improved to $O(1 + \mu)$ in expectation by the *Dynamic Parameterized Subset Sampling* (DPSS) technique [10] which was proposed to solve the problem defined as follows. Consider a set S containing n elements allowing element insertions or deletions. Each element $e \in S$ is associated with a *non-negative integer* weight $w(e)$. Given a pair of non-negative rational parameters (α, β), the objective is to return a subset of S such that each element $e \in S$ is independently sampled with probability $\min\left\{\frac{w(e)}{\alpha \sum_{e \in S} w(e) + \beta}, 1\right\}$. Such a subset sample can be obtained with DPSS in $O(1+\mu)$ expected time. To apply DPSS to the subset sampling for *AGP-Static++*, for each vertex u, the element set S is $N[u]$, where each vertex $v \in N[u]$

is an element with weight $w(v) = \lceil \frac{n_{\max}}{d_v^a} \rceil$, with $n_{\max}$ being the maximum possible n. A subset sample T from $N[u]$, where each $v \in N[u]$ is sampled independently with probability $p_{u,v}$, can be obtained by as follows:

- perform a DPSS query of $\alpha = 0$ and $\beta = \frac{n_{\max}}{s_u}$ to obtain a candidate set $S' \subseteq S$;
- for each $v \in S'$, accept v with probability $\frac{p_{u,v}}{w(v)/\beta} \in \Omega(1)$.

Denote the abobe implementation with DPSS of *AGP-Static++* by *AGP-DPSS*. While the subset sampling time for each $N[u]$ is bounded by $O(1+\mu_u)$ in expectation, where μ_u is the expected subset sample size in $N[u]$, *AGP-DPSS* cannot support the parameter a given on the fly as a is used to build the data structure, i.e., the weight of each element. Substituting $c_{\text{so}} = O(1)$ and $\varepsilon = O(\frac{\delta}{L})$ to Theorem 1, we have the following corollary:

Corollary 3. *The expected query time complexity of* AGP-DPSS *is bounded by*

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

4.2 Tighter Variance and Error Bound

We now analyze the variance of $\hat{\boldsymbol{\pi}}$ returned by our *AGP-Static++*. We define $\{\hat{\mathbf{r}}^{(l)}\}$ as a union of $\hat{\mathbf{r}}^{(i)}$ for $i = 0\cdots l$, and $p^{(i)}(u,v) = \mathbf{e}_v^T\cdot(\mathbf{D}^{-a}\mathbf{A}\mathbf{D}^{-b})^i\cdot\mathbf{e}_u$, which is the i^{th} normalized transition probability from vertex u to v, where $\mathbf{e}_v$ is a one-hot vector with $\mathbf{e}_v(v) = 1$. In particular, $p^{(0)}(u,v) = 1$ if and only if $u = v$; otherwise, $p^{(0)}(u,v) = 0$. Furthermore, $p^{(1)}(u,v) = \frac{1}{d_v^a\cdot d_u^b}$ if $(u,v)\in E$; otherwise, $p^{(1)}(u,v) = 0$. We first prove Lemma 1 and Lemma 2 regarding the expectation and the variance of $\hat{\mathbf{r}}^{(i)}$.

Lemma 1. $E\left[\hat{\mathbf{r}}^{(l)}(v)\middle|\{\hat{\mathbf{r}}^{(l-1)}\}\right] = \sum_{u\in N[v]}\frac{Y_l}{Y_{l-1}}\cdot p^{(1)}(u,v)\cdot\hat{\mathbf{r}}^{(l-1)}(u)$.

Proof. As according to Fact 2, $\hat{\boldsymbol{\pi}}(v) = \sum_{i=0}^{L}\hat{\mathbf{q}}^{(i)}(v) = \sum_{i=0}^{L}\frac{w_i}{Y_i}\cdot\hat{\mathbf{r}}^{(i)}(v)$ is an unbiased estimation, we have $\mathrm{E}[\hat{\boldsymbol{\pi}}(v)] = \sum_{i=0}^{L}\mathbf{q}^{(i)}(v)$.

First, we use $X^{(i)}(u,v)$ to denote the value propagated from u to v at step i, which is formally defined as below:

$$X^{(i)}(u,v) = \begin{cases}\frac{Y_i}{Y_{i-1}}\cdot\frac{\hat{\mathbf{r}}^{(i-1)}(u)}{d_v^a\cdot d_u^b} & \text{when } \frac{Y_i}{Y_{i-1}}\cdot\frac{\hat{\mathbf{r}}^{(i-1)}(u)}{d_v^a\cdot d_u^b}\geq\varepsilon\\ \begin{cases}\varepsilon & \text{w.p. } \frac{1}{\varepsilon}\cdot\frac{Y_i}{Y_{i-1}}\cdot\frac{\hat{\mathbf{r}}^{(i-1)}(u)}{d_v^a\cdot d_u^b}\\ 0 & \text{w.p. } 1-\frac{1}{\varepsilon}\cdot\frac{Y_i}{Y_{i-1}}\cdot\frac{\hat{\mathbf{r}}^{(i-1)}(u)}{d_v^a\cdot d_u^b}\end{cases} & \text{otherwise}\end{cases}$$

By the definition of $X^{(l)}(u,v)$, we have:
$E\left[X^{(l)}(u,v)\middle|\{\hat{\mathbf{r}}^{(l-1)}\}\right] = \frac{Y_l}{Y_{l-1}}\cdot\frac{\hat{\mathbf{r}}^{(l-1)}(u)}{d_v^a\cdot d_u^b}$. This is because, when $\frac{Y_i}{Y_{i-1}}\cdot\frac{\hat{\mathbf{r}}^{(i-1)}(u)}{d_v^a\cdot d_u^b}\geq\varepsilon$, $X^{(l)}(u,v) = \frac{Y_l}{Y_{l-1}}\cdot\frac{\hat{\mathbf{r}}^{(l-1)}(u)}{d_v^a\cdot d_u^b}$; otherwise
$E[X^{(l)}(u,v)] = \varepsilon\cdot\frac{1}{\varepsilon}\frac{Y_i}{Y_{i-1}}\cdot\frac{\hat{\mathbf{r}}^{(i-1)}(u)}{d_v^a\cdot d_u^b} + 0\cdot(1-\frac{1}{\varepsilon}\frac{Y_i}{Y_{i-1}}\cdot\frac{\hat{\mathbf{r}}^{(i-1)}(u)}{d_v^a\cdot d_u^b}) = \frac{Y_i}{Y_{i-1}}\cdot\frac{\hat{\mathbf{r}}^{(i-1)}(u)}{d_v^a\cdot d_u^b}$. As $X^{(l)}(u,v)$ denotes the value propagated from u to v at step l, hence $\hat{\mathbf{r}}^{(l)}(v) = \sum_{u\in N[v]}X^{(l)}(u,v)$. Therefore, we have $E\left[\hat{\mathbf{r}}^{(l)}(v)\middle|\{\hat{\mathbf{r}}^{(l-1)}\}\right] = E\left[\sum_{u\in N[v]}X^{(l)}(u,v)\middle|\{\hat{\mathbf{r}}^{(l-1)}\}\right] = \sum_{u\in N[v]}\frac{Y_l}{Y_{l-1}}\cdot\frac{\hat{\mathbf{r}}^{(l-1)}(u)}{d_v^a\cdot d_u^b}$. □

Lemma 2. $Var\left[\hat{\mathbf{r}}^{(l)}(v)\middle|\{\hat{\mathbf{r}}^{(l-1)}\}\right] \leq \sum_{u\in N[v]}\varepsilon\cdot\frac{Y_l}{Y_{l-1}}\cdot p^{(1)}(u,v)\cdot\hat{\mathbf{r}}^{(l-1)}(u)$.

Proof. Observe that when $\frac{Y_l}{Y_{l-1}}\cdot\frac{\hat{\mathbf{r}}^{(l-1)}(u)}{d_v^a\cdot d_u^b}\geq\varepsilon$, $X^{(l)}(u,v)$ is deterministic, hence this case has no variance. It thus suffices to focus on the case that $\frac{Y_l}{Y_{l-1}}\cdot\frac{\hat{\mathbf{r}}^{(l-1)}(u)}{d_v^a\cdot d_u^b} < \varepsilon$, where $X^{(l)}(u,v)\leq\varepsilon$. We have $\mathrm{Var}\left[X^{(l)}(u,v)\middle|\{\hat{\mathbf{r}}^{(l-1)}\}\right] = \mathrm{E}\left[\left(X^{(l)}(u,v)\right)^2\middle|\{\hat{\mathbf{r}}^{(l-1)}\}\right] - \left(\mathrm{E}\left[X^{(l)}(u,v)\middle|\{\hat{\mathbf{r}}^{(l-1)}\}\right]\right)^2 \leq \mathrm{E}\left[\left(X^{(l)}(u,v)\right)^2\middle|\{\hat{\mathbf{r}}^{(l-1)}\}\right] \leq \varepsilon^2\cdot\frac{1}{\varepsilon}\cdot\frac{Y_l}{Y_{l-1}}\cdot\frac{\hat{\mathbf{r}}^{(l-1)}(u)}{d_v^a\cdot d_u^b} = \varepsilon\cdot\frac{Y_l}{Y_{l-1}}\cdot\frac{\hat{\mathbf{r}}^{(l-1)}(u)}{d_v^a\cdot d_u^b}$. Again, by the fact that $\hat{\mathbf{r}}^{(l)}(v) = \sum_{u\in N[v]}X^{(l)}(u,v)$, we then have $\mathrm{Var}\left[\hat{\mathbf{r}}^{(l)}(v)\middle|\{\hat{\mathbf{r}}^{(l-1)}\}\right] = \mathrm{Var}\left[\sum_{u\in N[v]}X^{(l)}(u,v)\middle|\{\hat{\mathbf{r}}^{(l-1)}\}\right] \leq \sum_{u\in N[v]}\varepsilon\cdot\frac{Y_l}{Y_{l-1}}\cdot\frac{\hat{\mathbf{r}}^{(l-1)}(u)}{d_v^a\cdot d_u^b} = \sum_{u\in N[v]}\varepsilon\cdot\frac{Y_l}{Y_{l-1}}\cdot p^{(1)}(u,v)\cdot\hat{\mathbf{r}}^{(l-1)}(u)$. □

Lemma 3. *For $k = 0\ldots L-1$, we have:*

$$\begin{aligned}&Var\left[\sum_{j=0}^{k}\frac{w_{L-k+j}}{Y_{L-k}}\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-k)}(u) + \sum_{i=0}^{L-k-1}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right]\\ &= \varepsilon\cdot\sum_{j=0}^{k}\mathbf{q}^{(L-k+j)}(v) + Var\left[\sum_{j=0}^{k+1}\frac{w_{L-(k+1)+j}}{Y_{L-(k+1)}}\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-(k+1))}(u)\right.\\ &\qquad\left. + \sum_{i=0}^{L-(k+1)-1}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right].\end{aligned}$$

Next, we prove Lemma 3 by **mathematical induction**. We first prove the **base case** where $k = 0$, which is Lemma 4.

Lemma 4.

$$\begin{aligned}&\underbrace{Var\left[\sum_{j=0}^{0}\frac{w_{L+j}}{Y_L}\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L)}(u) + \sum_{i=0}^{L-1}\frac{w_i}{Y_i}\cdot\hat{\mathbf{r}}^{(i)}(v)\right]}_{(2)}\\ &= \varepsilon\cdot\mathbf{q}^{(L)}(v) + Var\left[\sum_{j=0}^{1}\frac{w_{L-1+j}}{Y_{L-1}}\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-1)}(u)\right.\\ &\qquad\left. + \sum_{i=0}^{L-2}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right]\end{aligned}$$

Proof. By the law of Total Variance on Term (2), we have:

$$\begin{aligned}\text{Term (2)} &= \mathrm{Var}\left[\sum_{i=0}^{L}\frac{w_i}{Y_i}\cdot\hat{\mathbf{r}}^{(i)}(v)\right]\\ &= \underbrace{\mathrm{E}\left[\mathrm{Var}\left[\sum_{i=0}^{L}\frac{w_i}{Y_i}\cdot\hat{\mathbf{r}}^{(i)}(v)\middle|\{\hat{\mathbf{r}}^{(L-1)}\}\right]\right]}_{(3)}\\ &\quad + \underbrace{\mathrm{Var}\left[\mathrm{E}\left[\sum_{i=0}^{L}\frac{w_i}{Y_i}\cdot\hat{\mathbf{r}}^{(i)}(v)\middle|\{\hat{\mathbf{r}}^{(L-1)}\}\right]\right]}_{(4)}.\end{aligned}$$

We first look into Term (3). As $\{\hat{\mathbf{r}}^{(L-1)}\}$ is given, the variance only comes from $\hat{\mathbf{r}}^{(L)}$. By Lemma 2 and the fact that $\hat{\mathbf{r}}$ is an unbiased estimation of $\mathbf{r}$:

$$\begin{aligned}\text{Term (3)} &= \mathrm{E}\left[\mathrm{Var}\left[\frac{w_L}{Y_L}\cdot\hat{\mathbf{r}}^{(L)}(u)\middle|\{\hat{\mathbf{r}}^{(L-1)}\}\right]\right]\\ &\leq \mathrm{E}\left[(\frac{w_L}{Y_L})^2\cdot\varepsilon\cdot\hat{\mathbf{r}}^{(L)}(v)\right] \leq \varepsilon\cdot\frac{w_L}{Y_L}\cdot\mathbf{r}^{(L)}(v) = \varepsilon\cdot\mathbf{q}^{(L)}(v),\end{aligned}$$

where the last inequality follows from $w_L \le \sum_{j=L}^{\infty} w_j = Y_L$.

Next, we analyze Term (4), again as $\left\{\hat{\mathbf{r}}^{(L-1)}\right\}$ is given, and by Lemma 1:

$$\begin{aligned}
\text{Term (4)} &= \mathrm{Var}\left[\mathrm{E}\left[\frac{w_L}{Y_L}\cdot\hat{\mathbf{r}}^{(L)}(v)+\sum_{i=0}^{L-1}\frac{w_i}{Y_i}\cdot\hat{\mathbf{r}}^{(i)}(v)\middle|\left\{\hat{\mathbf{r}}^{(L-1)}\right\}\right]\right]\\
&= \mathrm{Var}\left[\mathrm{E}\left[\frac{w_L}{Y_L}\cdot\hat{\mathbf{r}}^{(L)}(v)\middle|\left\{\hat{\mathbf{r}}^{(L-1)}\right\}\right]+\sum_{i=0}^{L-1}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right]\\
&= \mathrm{Var}\left[\frac{w_L}{Y_L}\sum_{u\in N[v]}\frac{Y_L}{Y_{L-1}}\cdot p^{(1)}(u,v)\cdot\hat{\mathbf{r}}^{(L-1)}(u)+\sum_{i=0}^{L-1}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right]\\
&= \mathrm{Var}\Bigg[\sum_{u\in N[v]}\frac{w_L}{Y_{L-1}}\cdot p^{(1)}(u,v)\cdot\hat{\mathbf{r}}^{(L-1)}(u)+\frac{w_{L-1}}{Y_{L-1}}\hat{\mathbf{r}}^{(L-1)}(v)\\
&\qquad+\sum_{i=0}^{L-2}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\Bigg].
\end{aligned}$$

By the definition of $p^{(j)}(u,v)$, Term (4) can be rewritten as: $\mathrm{Var}\left[\sum_{j=0}^{1}\frac{w_{L-1+j}}{Y_{L-1}}\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-1)}(u)+\sum_{i=0}^{L-2}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right]$.

Summing Term (3) and Term (4) up, Lemma 4 follows. □

We then prove the **inductive case** as stated in Lemma 5.

Lemma 5. *If Lemma 3 holds for $k = l$, then it is also true for $k = l+1$ for $0 \le l < L$.*

Proof. When $k = l$, by Lemma 3 we have

$$\begin{aligned}
&\mathrm{Var}\left[\sum_{j=0}^{l}\frac{w_{L-l+j}}{Y_{L-l}}\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-l)}(u)+\sum_{i=0}^{L-l-1}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right]\\
&=\varepsilon\cdot\sum_{j=0}^{l}\mathbf{q}^{(L-l+j)}(v)\\
&\quad+\underbrace{\mathrm{Var}\left[\sum_{j=0}^{l+1}\frac{w_{L-(l+1)+j}}{Y_{L-(l+1)}}\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-(l+1))}(u)+\sum_{i=0}^{L-(l+1)-1}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right]}_{(5)}.
\end{aligned}$$

Again, we apply the law of Total Variance on Term (5):

$$\begin{aligned}
&\text{Term (5)}\\
&=\underbrace{\mathrm{E}\left[\mathrm{Var}\left[\sum_{j=0}^{l+1}\frac{w_{L-(l+1)+j}}{Y_{L-(l+1)}}\cdot\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-(l+1))}(u)+\sum_{i=0}^{L-(l+1)-1}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\middle|\left\{\hat{\mathbf{r}}^{(L-(l+1)-1)}\right\}\right]\right]}_{(6)}\\
&\quad+\underbrace{\mathrm{Var}\left[\mathrm{E}\left[\sum_{j=0}^{l+1}\frac{w_{L-(l+1)+j}}{Y_{L-(l+1)}}\cdot\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-(l+1))}(u)+\sum_{i=0}^{L-(l+1)-1}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\middle|\left\{\hat{\mathbf{r}}^{(L-(l+1)-1)}\right\}\right]\right]}_{(7)}.
\end{aligned}$$

We first analyze Term (6). As $\left\{\hat{\mathbf{r}}^{(L-(l+1)-1)}\right\}$ is given, hence

$$\begin{aligned}
&\text{Term (6)}\\
&=\mathrm{E}\left[\mathrm{Var}\left[\sum_{j=0}^{l+1}\frac{w_{L-(l+1)+j}}{Y_{L-(l+1)}}\cdot\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-(l+1))}(u)\middle|\left\{\hat{\mathbf{r}}^{(L-(l+1)-1)}\right\}\right]\right].
\end{aligned}$$

By Lemma 2, $\mathrm{Var}\left[\hat{\mathbf{r}}^{(L-(l+1))}(u)\middle|\left\{\hat{\mathbf{r}}^{(L-(l+1)-1)}\right\}\right] \le \sum_{w\in N[u]}\varepsilon\cdot\frac{Y_{L-(l+1)}}{Y_{L-(l+1)-1}}\cdot p^{(1)}(w,u)\cdot\hat{\mathbf{r}}^{(L-(l+1)-1)}(w)$, and by the definition of variance, we can further have:

$$\begin{aligned}
\text{Term (6)} \le \mathrm{E}\Bigg[&\sum_{u\in V}\left(\sum_{j=0}^{l+1}\frac{w_{L-(l+1)+j}}{Y_{L-(l+1)}}\cdot p^{(j)}(u,v)\right)^2\cdot\sum_{w\in N[u]}\varepsilon\cdot\frac{Y_{L-(l+1)}}{Y_{L-(l+1)-1}}\\
&\cdot p^{(1)}(w,u)\cdot\hat{\mathbf{r}}^{(L-(l+1)-1)}(w)\Bigg].
\end{aligned}$$

By the fact that $p^{(j)}(u,v) \le 1$ and $\sum_{j=0}^{l+1}\frac{w_{L-(l+1)+j}}{Y_{L-(l+1)}} \le \frac{Y_{L-(l+1)}}{Y_{L-(l+1)}} \le 1$, Term (6) can be bounded as:

$$\begin{aligned}
\text{Term (6)} &\le \mathrm{E}\Bigg[\varepsilon\cdot\sum_{j=0}^{l+1}\frac{w_{L-(l+1)+j}}{Y_{L-(l+1)}}\cdot\sum_{u\in V}p^{(j)}(u,v)\cdot\sum_{w\in N[u]}\frac{Y_{L-(l+1)}}{Y_{L-(l+1)-1}}\\
&\qquad\cdot p^{(1)}(w,u)\cdot\hat{\mathbf{r}}^{(L-(l+1)-1)}(w)\Bigg]\\
&= \mathrm{E}\left[\varepsilon\cdot\sum_{j=0}^{l+1}\frac{w_{L-(l+1)+j}}{Y_{L-(l+1)}}\cdot\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-(l+1))}(u)\right].
\end{aligned}$$

According to the definition of $\hat{\mathbf{r}}^{(i)}(v)$, we have: $\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-(l+1))}(u) = \hat{\mathbf{r}}^{(L-(l+1)+j)}(v)\cdot\frac{Y_{L-(l+1)}}{Y_{L-(l+1)+j}}$. Intuitively, it processes j more steps of propagation to v. As a result, Term (6) $\le \varepsilon\cdot\sum_{j=0}^{l+1}\mathbf{q}^{(L-(l+1)+j)}(v)$.

We now consider Term (7). Again, as $\left\{\hat{\mathbf{r}}^{(L-(l+1)-1)}\right\}$ is given, and by Lemma 1,

$$\begin{aligned}
&\text{Term (7)}\\
&=\mathrm{Var}\left[\mathrm{E}\left[\sum_{j=0}^{l+1}\frac{w_{L-(l+1)+j}}{Y_{L-(l+1)}}\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-(l+1))}(u)\middle|\left\{\hat{\mathbf{r}}^{(L-(l+1)-1)}\right\}\right]\right.\\
&\qquad\left.+\sum_{i=0}^{L-(l+1)-1}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right]\\
&=\mathrm{Var}\left[\sum_{j=0}^{l+1}\frac{w_{L-(l+1)+j}}{Y_{L-(l+1)}}\sum_{u\in V}p^{(j)}(u,v)\sum_{w\in N[u]}\frac{Y_{L-(l+1)}}{Y_{L-(l+1)-1}}\cdot p^{(1)}(w,u)\right.\\
&\qquad\left.\cdot\hat{\mathbf{r}}^{(L-(l+1)-1)}(w)+\sum_{i=0}^{L-(l+1)-1}\frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right].
\end{aligned}$$

With $\sum_{w\in N[u]}\frac{Y_{L-(l+1)}}{Y_{L-(l+1)-1}}\cdot p^{(1)}(w,u)\cdot\hat{\mathbf{r}}^{(L-(l+1)-1)}(w) = \hat{\mathbf{r}}^{(L-(l+1))}(u)$, we can rewrite $\sum_{j=0}^{l+1}\frac{w_{L-(l+1)+j}}{Y_{L-(l+1)}}\sum_{u\in V}p^{(j)}(u,v)\sum_{w\in N[u]}\frac{Y_{L-(l+1)}}{Y_{L-(l+1)-1}}\cdot p^{(1)}(w,u)\cdot\hat{\mathbf{r}}^{(L-(l+1)-1)}(w)$ as $\sum_{j=1}^{l+2}\frac{w_{L-(l+2)+j}}{Y_{L-(l+2)}}\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-(l+2))}(u)$. We can then split $\frac{w_{L-(l+1)-1}}{Y_{L-(l+1)-1}}\hat{\mathbf{r}}^{(L-(l+1)-1)}(v) = \sum_{j=0}^{0}\frac{w_{L-(l+2)+j}}{Y_{L-(l+2)}}\sum_{u\in V}p^{(j)}(u,v)\cdot\hat{\mathbf{r}}^{(L-(l+2))}(u)$ from the right

term, $\sum_{i=0}^{L-(l+1)-1} \frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)$, and adds it to the left term to get:

$$\text{Term (7)} = \text{Var}\left[\sum_{j=0}^{l+2} \frac{w_{L-(l+2)+j}}{Y_{L-(l+2)}} \sum_{u\in V} p^{(j)}(u,v)\cdot \hat{\mathbf{r}}^{(L-(l+2))}(u) + \sum_{i=0}^{L-(l+2)-1} \frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right].$$

Therefore, we have Term (5) = Term (6) + Term (7)

$\leq \varepsilon \cdot \sum_{j=0}^{l+1} \mathbf{q}^{(L-(l+1)+j)}(v) + \text{Var}\left[\sum_{j=0}^{l+2} \frac{w_{L-(l+2)+j}}{Y_{L-(l+2)}} \sum_{u\in V} p^{(j)}(u,v)\cdot \hat{\mathbf{r}}^{(L-(l+2))}(u) + \sum_{i=0}^{L-(l+2)-1} \frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right]$.

This completes the proof of Lemma 5. □

By Lemma 4 (the base case when $k = 0$) and Lemma 5 (the inductive case), the mathmetical induction proof for Lemma 3 establishes. Finally, by Lemma 3, we can bound the variance:

$$\begin{aligned}
\text{Var}[\hat{\boldsymbol{\pi}}(v)] &= \text{Var}\left[\sum_{i=0}^{L} \frac{w_i}{Y_i}\cdot \hat{\mathbf{r}}^{(i)}(v)\right] \\
&= \text{Var}\left[\sum_{j=0}^{0} \frac{w_{L+j}}{Y_L} \sum_{u\in V} p^{(j)}(u,v)\hat{\mathbf{r}}^{(L)}(u) + \sum_{i=0}^{L-1} \frac{w_i}{Y_i}\hat{\mathbf{r}}^{(i)}(v)\right] \\
&= \varepsilon \sum_{j=0}^{0} \mathbf{q}^{(L-1+j)}(v) + \cdots + \varepsilon \sum_{j=0}^{L-1} \mathbf{q}^{(1+j)}(v) \\
&\quad + \text{Var}\left[\sum_{j=0}^{L} w_j \sum_{u\in V} p^{(j)}(u,v)\hat{\mathbf{r}}^{(0)}(u)\right] \\
&= \varepsilon \sum_{k=0}^{L}\sum_{j=0}^{k} \mathbf{q}^{(L-k+j)}(v) + \text{Var}\left[\sum_{j=0}^{L} w_j \sum_{u\in V} p^{(j)}(u,v)\hat{\mathbf{r}}^{(0)}(u)\right].
\end{aligned}$$

Since $\varepsilon\cdot\sum_{k=0}^{L}\sum_{j=0}^{k} \mathbf{q}^{(L-k+j)}(v) \leq \varepsilon\cdot L\sum_{j=0}^{L}\mathbf{q}^{(j)}(v) \leq \varepsilon\cdot L\cdot\boldsymbol{\pi}(v)$. And by the definition of variance, we have:

$$\text{Var}\left[\sum_{j=0}^{L} w_j \cdot \sum_{u\in V} p^{(j)}(u,v)\cdot \hat{\mathbf{r}}^{(0)}(u)\right] = \sum_{u\in V}\left(\sum_{j=0}^{L} w_j \cdot p^{(j)}(u,v)\right)^2 \cdot \text{Var}[\hat{\mathbf{r}}^{(0)}(u)].$$

Therefore,

$$\text{Var}[\hat{\boldsymbol{\pi}}(v)] \leq \varepsilon\cdot L\cdot\boldsymbol{\pi}(v) + \sum_{u\in V}\left(\sum_{j=0}^{L} w_j \cdot p^{(j)}(u,v)\right)^2 \cdot \text{Var}[\hat{\mathbf{r}}^{(0)}(u)].$$

Here, $\text{Var}[\hat{\mathbf{r}}^{(0)}(u)]$ is related to the initialization method. When $\hat{\mathbf{r}}^{(0)}$ is initialized deterministically (e.g., Line 1 in Algorithm 2), $\text{Var}[\hat{\boldsymbol{r}}^{(0)}(u)] = 0$, and thus, $\text{Var}[\hat{\boldsymbol{\pi}}(v)] \leq \varepsilon\cdot L\cdot\boldsymbol{\pi}(v)$.

Moreover, in Section 6, we propose a randomized initialization algorithm for some special cases when the input $\mathbf{x}$ satisfies certain conditions. We prove that, with our randomized initialization, $\text{Var}[\hat{\boldsymbol{\pi}}(v)] \leq \varepsilon\cdot(L+1)\cdot\boldsymbol{\pi}(v)$ holds.

Therefore, in either initialization way, by Chebyshev's Inequality [28], when $\varepsilon = \frac{c^2\delta}{2(L+1)} = O(\frac{\delta}{L})$, we have:

$$\Pr[|\boldsymbol{\pi}(v) - \hat{\boldsymbol{\pi}}(v)| > c\cdot\boldsymbol{\pi}(v)] \leq \frac{\text{Var}[\hat{\boldsymbol{\pi}}(v)]}{c^2\cdot\boldsymbol{\pi}(v)^2} \leq \frac{\varepsilon(L+1)}{c^2\cdot\boldsymbol{\pi}(v)} \leq \frac{\varepsilon(L+1)}{c^2\delta} = O(1).$$

Together with Fact 1 and Algorithm 2, the correctness of our *AGP-Static++* follows and is summarized in the theorem below:

Theorem 2. *For any query $q(a,b,\mathcal{O}_w,\mathbf{x})$, our* AGP-Static++ *can return an (δ,c)-approximation $\hat{\boldsymbol{\pi}}$ with at least a constant probability.*

5 AGP ON DYNAMIC GRAPHS

While our *AGP-Static++* can achieve good query time complexity, it does not work well with graph updates. To see this, suppose there is an update (either an insertion or a deletion) of an edge (u,w); due to this update, the degree d_u (resp., d_w) changes, and hence, the sampling probabilities of u within the neighborhoods $N[v]$ of all u's neighbors v change accordingly. As a result, while the subset sampling structures in *AGP-Static++* can be updated efficiently in $O(1)$ time, the overall cost for each graph update incurs $O(d_u)$ (resp., $O(d_w)$) time, which can be as large as $O(n)$ in the worst case. This limits the applications of *AGP-Static++* to scenarios where the underlying graph G is updated frequently.

To address this challenge, we introduce a strengthened version of *AGP-Static++*, called *AGP-Dynamic*, which can support each edge insertion or deletion in $O(1)$ amortized time (substantially improving the aforementioned $O(n)$ update time bound). The core idea of *AGP-Dynamic* is an observation that it is indeed *unnecessary* to update the sampling probability for u within the neighborhoods of u's neighbors for *every* graph update related to u, as long as our algorithm can still have a *good overestimate* p^* of $p_{v,u}$ such that $p_{\text{ac}} = \frac{p_{v,u}}{p^*} \in \Omega(1)$. If this is the case, one can still first sample u with probability p^* and set the accept probability p_{ac} with respect to u's current degree. According to our theoretical analysis, the subset sampling complexity within each neighborhood would remain the same, and so as the overall expected query time bound.

Performing Updates. To achieve this, *AGP-Dynamic* maintains a *reference degree*, denoted by $\tilde{d}_u$, for each vertex $u \in V$, which records the degree value of u when the *last* update for u within the neighborhoods of u's neighbors, and initially, $\tilde{d}_u = d_u$. The detailed implementation is shown in Algorithm 3.

Algorithm 3: *AGP-Dynamic*

```
Input: an edge update (u, w)
1 if (u, w) is a deletion then remove u from B[w] and d_u ← d_u − 1;
2 else insert u to B[w] and d_u ← d_u + 1;
3 if d_u < (1/2) · d̃_u or d_u > 2 · d̃_u then
4     for each v ∈ N[u] do
5         update B[v] based on current d_u;
6     d̃_u ← d_u;
7 handle w symmetrically
```

Theorem 3. *Algorithm 3 can process each graph update in $O(1)$ amortized time.*

Proof. It is known that, the sorted linked list $B[u]$ can be maintained using $O(|B[u]|)$ space, and supports each insertion or deletion of a bucket into or from $B[u]$ in $O(1)$ time [10]. With that, removing or inserting an element in $B[u]$ (Lines 1 and 2) takes $O(1)$. Therefore, updating u in $B[v]$ within the neighborhoods of u's neighbors (Lines 4-5) takes $O(d_u)$ time. However, according to the condition in Line 3, such an update cost only incurs when there are $\Omega(d_u)$ updates related to u occurred, because, otherwise, d_u would not have been changed more than a factor of 2 from $\tilde{d}_u$. As a result, this $O(d_u)$ update cost can be charged to these $\Omega(d_u)$ updates, and thus, the amortized update cost becomes $O(1)$. □

Performing Queries. Observe that subset sampling structure within the neighborhood $N[v]$ for each $v \in V$ is built on the reference degrees $\tilde{d}_u$ of v's neighbors $u \in N[v]$. As a result, the previous overestimate probability p^* of $p_{v,u}$ used in *AGP-Static++*, which is computed based on the bucket index storing u in $B[v]$, is no longer guaranteed to be feasible. This is because the current degree d_u can now be smaller than $\tilde{d}_u$, making the actual sampling probability $p_{v,u} > p^*$. Fortunately, one crucial observation here is that, according to Line 3 in Algorithm 3, $d_u \geq \frac{1}{2}\tilde{d}_u$ always holds. Therefore, $p_{v,u} \leq 2^a \cdot p^*$ holds, and thus, $p^{*\prime} = 2^a \cdot p^*$ can be used as a good overestimate of $p_{v,u}$ for the subset sampling within $N[v]$ for u. Hence, the correctness of the query result of *AGP-Dynamic* is guaranteed. Moreover, as discussed earlier, the expected query time complexity of *AGP-Dynamic* is the same as that of *AGP-Static++*.

6 RANDOMIZED INITIALIZATION

In *AGP-Static*, $\hat{\mathbf{r}}^{(0)}$ is initialized to $\mathbf{x}$ taking $O(n)$ time, which is not guaranteed to be dominated by $c_{\text{so}} \cdot \frac{1}{\varepsilon} \cdot \sum_{i=1}^{L} \left\| Y_i (D^{-a} A D^{-b})^i \mathbf{x} \right\|_1$. In certain applications, $\mathbf{x}$ admits a *compact representation*, making a full read unnecessary. A notable example is the PageRank [27]. We exploit this property and propose a sub-linear time randomized algorithm for initialization, when $\mathbf{x}$ satisfies Condition 1.

Condition 1. *$\mathbf{x}$ satisfies following conditions and can be represented as P and S:*

- *$\mathbf{x}$ can be partitioned into $O(\frac{1}{\varepsilon})$ index groups denoted as P, where $P_i \subseteq \{1, 2, \cdots, n\}$ indicates the i^{th} group;*
- *for each P_i, the indexes are continuative;*
- *for any two indexes $j, k \in P_i$, we have $\mathbf{x}_j = \mathbf{x}_k$ stored at S_i.*

As shown in Algorithm 4, our randomized initialization algorithm works as follows. When $\frac{\mathbf{x}(v)}{\varepsilon} \geq 1$, the exact value of $\mathbf{x}(v)$ is assigned to $\hat{\mathbf{r}}^{(0)}(v)$. Otherwise, subset sampling with uniform probability is performed on partition P_i, where each index is sampled with probability $\frac{S_i}{\varepsilon}$. Once an index j is selected, ε is assigned to $\mathbf{x}_j$. Trivially, this algorithm assures that $E[\hat{\mathbf{r}}^{(0)}(v)] = \mathbf{x}(v)$ for any $v \in V$, while it introduces additional variance to $Var[\hat{\boldsymbol{\pi}}]$ for the whole algorithm. However, as will be shown next, this does not affect the overall error bound and query complexity bound of *AGP-Static++*.

Algorithm 4: *Randomized Initialization*

Input: P, S, and a parameter ε
Output: $\hat{\mathbf{r}}^{(0)}$

```
for i = 0 to |P| do
    if S_i/ε ≥ 1 then
        for j ∈ P_i do
            r̂^(0)_j ← S_i;
    else
        p = S_i/ε;
        K = SubsetSample(p, |P|);
        for j ∈ K do
            r̂^(0)_j ← ε;
return r̂^(0)
```

Table 2: Dataset Statistics

Datasets	n ($\times 10^6$)	m ($\times 10^6$)	$\bar{d}$	Domain
soc-Slashdot0811 (**Sl**)	0.08	0.47	12.13	Social network
web-NotreDame (**ND**)	0.33	1.09	6.69	Website hyperlink
web-Google (**Go**)	0.88	4.32	9.86	Website hyperlink
wiki-Topcats (**To**)	1.79	25.44	28.38	Website hyperlink
soc-Pokec (**Po**)	1.63	22.30	27.36	Social network
as-Skitter (**Sk**)	1.70	11.10	13.06	Traceroute graph
wiki-Talk (**Ta**)	2.39	4.66	3.90	Interaction graph
soc-Orkut (**Or**)	3.07	117.19	76.22	Social network
soc-LiveJournal1 (**LJ**)	4.85	42.85	17.69	Social Network

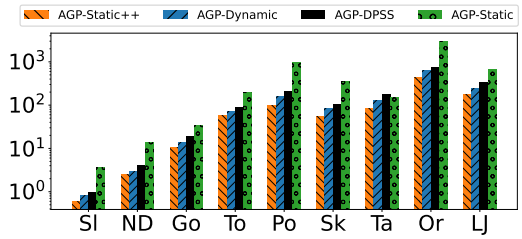


Figure 1: Average query processing time (second)

Variance Analysis. When Algorithm 4 is applied, the variance comes from only when $\frac{\mathbf{x}(u)}{\varepsilon} < 1$, as it is a deterministic term when $\frac{\mathbf{x}(u)}{\varepsilon} \geq 1$. Therefore, $\text{Var}[\hat{\mathbf{r}}^{(0)}(u)] \leq E[(\hat{\mathbf{r}}^{(0)})^2] \leq \varepsilon \cdot \mathbf{r}^{(0)}(u)$, where $E[\hat{\mathbf{r}}^{(0)}(u)] = \mathbf{x}(u)$. Moreover, since $\Sigma_{j=0}^{L} w_j \leq 1$ and $p^{(j)}(u, v) \leq 1$, we have

$$\sum_{u \in V} \left(\sum_{j=0}^{L} w_j \cdot p^{(j)}(u, v) \right)^2 \cdot \text{Var}[\hat{\mathbf{r}}^{(0)}(u)]$$

$$\leq \varepsilon \cdot \sum_{u \in V} \left(\sum_{j=0}^{L} w_j \cdot p^{(j)}(u, v) \right) \cdot \mathbf{r}^{(0)}(u) = \varepsilon \cdot \boldsymbol{\pi}(v)$$

Complexity Analysis. There are at most $O(\frac{1}{\varepsilon})$ number of values in $\mathbf{x}$ where $\frac{\mathbf{x}(v)}{\varepsilon} \geq 1$, otherwise their sum already exceeds 1 – recall that $\|\mathbf{x}\|_1 = 1$. When running the subset sampling for each partition, the expected time complexity is $O(\mu_i + 1)$, where μ_i is the expected output size, and we have $\mu_i = |P_i| \cdot \frac{S_i}{\varepsilon}$. Hence, the total expected time complexity for all buckets is bounded by $O(\sum_i \mu_i + \frac{1}{\varepsilon}) = O(\frac{1}{\varepsilon} \cdot (\sum_i |P_i| \cdot S_i + 1)) = O(\frac{1}{\varepsilon})$. Therefore, the complexity of Algorithm 4 is bounded by $O(\frac{1}{\varepsilon})$.

7 EXPERIMENTS

Datasets. We use nine real-world datasets [22], soc-Slashdot0811 (**Sl**), web-NotreDame (**ND**), web-Google (**Go**), wiki-Topcats (**To**), soc-Pokec (**Po**), as-Skitter (**Sk**), wiki-Talk (**Ta**), soc-Orkut (**Or**), and soc-LiveJournal1 (**LJ**) with up to 4.8 million vertices and 117 million edges, whose statistics can be found in Table 2. We focus on query, update, and initialization efficiency, which are detailed next. Our source code can be found online [36].

7.1 Query Efficiency

Baseline. We evaluate our algorithms, *AGP-Static++* and *AGP-Dynamic*, against *AGP-Static* [32] and *AGP-DPSS* (Section 4.1).

Parameter Setting. We evaluate all applications in Table 1 using their respective parameter settings. Following [32], we set $\alpha = 0.2$ for random walk, $t = 4$ for Poisson distribution. For applications where $\mathbf{x}$ is a one-hot vector, we construct it by selecting j uniformly at random from $[1, n]$ and setting $\mathbf{x}_j = 1$. For GNN applications,

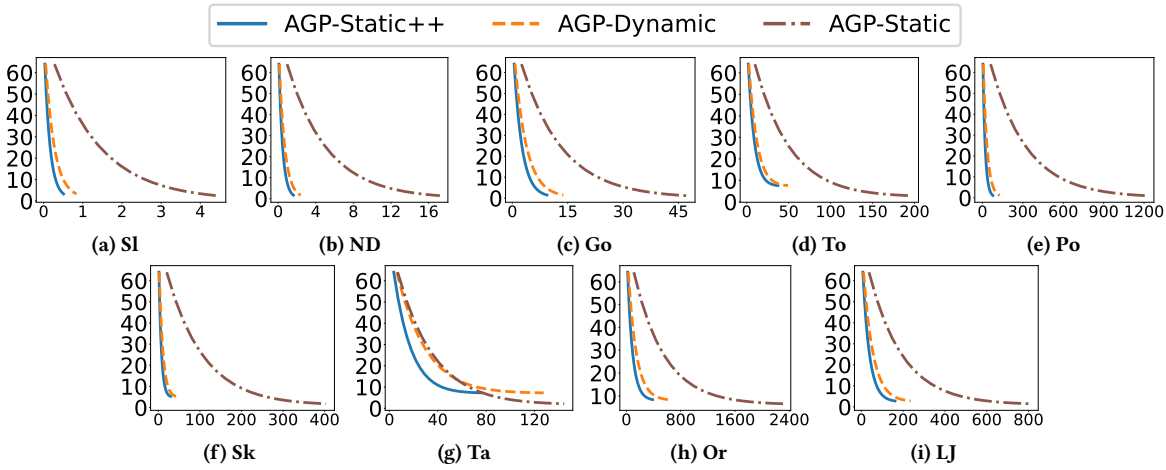


Figure 2: Relative error (%) vs. query time (second)

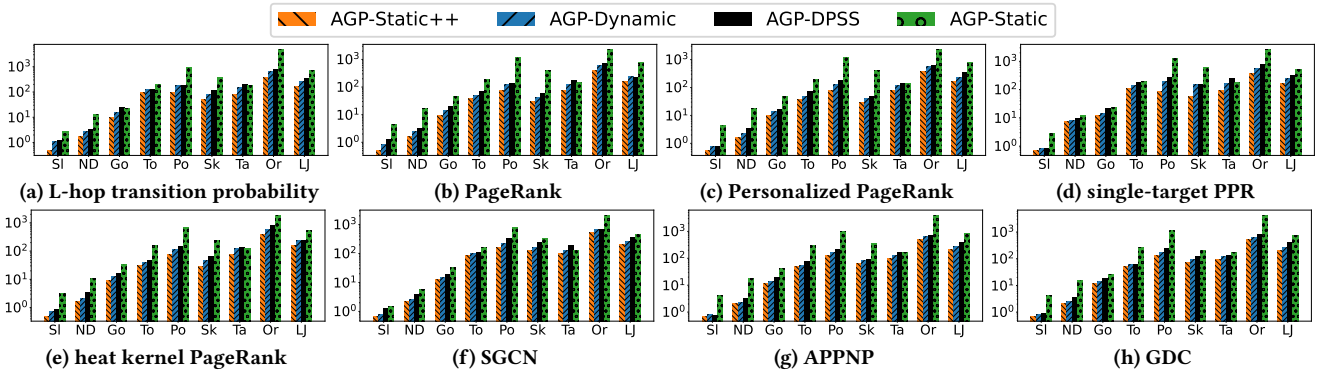


Figure 3: Average query time (second) on different applications

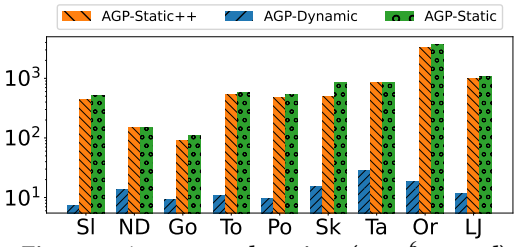


Figure 4: Average update time ($\times 10^{-6}$ second)

Table 3: Comparison of Query Time, Average Update Time, and Average Number of Updates per Query Threshold

	Query time		Average update time ($\times 10^{-3}$)		updates/query
	AGP-Static++	*AGP-Dynamic*	*AGP-Static++*	*AGP-Dynamic*	
Sl ($\bar{d}$ = 12.13)	0.596	0.827	0.51	0.01	461
ND ($\bar{d}$ = 6.69)	2.493	3.009	0.35	0.03	1617
Go ($\bar{d}$ = 9.86)	10.633	13.854	1.02	0.09	3459
To ($\bar{d}$ = 28.38)	56.823	70.018	30.93	0.58	435
Po ($\bar{d}$ = 27.36)	100.400	157.439	25.40	0.47	2287
Sk ($\bar{d}$ = 13.06)	53.745	82.031	19.79	0.36	1455
Ta ($\bar{d}$ = 3.9)	85.588	130.488	8.58	0.28	5412
Or ($\bar{d}$ = 76.22)	427.653	614.773	913.36	4.67	206
LJ ($\bar{d}$ = 17.69)	176.794	248.747	96.62	1.08	753

we generate n values following a uniform distribution with their sum equal to 1 and assign them to $\mathbf{x}$. We set $\delta = \frac{1}{n}$, $c = 0.1$ and $L = O(\log n)$ for all queries.

Results. For each dataset, we run five queries using five randomly generated instances of $\mathbf{x}$, except for PageRank, where $\mathbf{x} = [1/n, \ldots, 1/n]^T$ remains fixed. The average runtime of all applications on each dataset is reported in Figure 1. As discussed in Section 4.1, our tighter bounds lead to improvements in query time across all datasets. Notably, on Po, our *AGP-Static++* speeds up *AGP-Static* by 9.74×. For *AGP-Dynamic*, it picks a larger p^* in subset sampling, and hence generates a larger candidate sample set for rejection sampling. Nonetheless, it still outperforms *AGP-Static* by up to 6.21× (also on Po). Meanwhile, despite its theoretical superiority, *AGP-DPSS* performs worse than *AGP-Static++*; this is mainly due to the larger hidden constants introduced by the more complex structure of DPSS. The results for each individual application demonstrate similar results, which are shown in Figure 3. Across all applications, our method *AGP-Static++* achieves the best performance, thanks to its tighter bound and the new subset sampling algorithm. *AGP-Dynamic* exhibits slightly higher query time due to its use of a larger p^* in each bucket. *AGP-DPSS* performs slightly worse, primarily because of the large hidden constant in its query complexity.

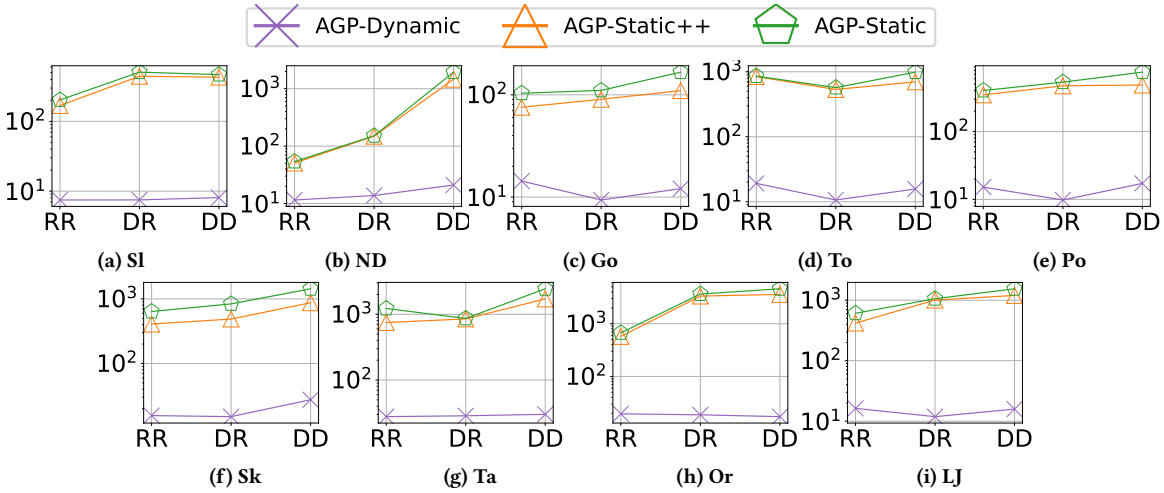


Figure 5: Average update time ($\times 10^{-6}$ sec) vs. update pattern

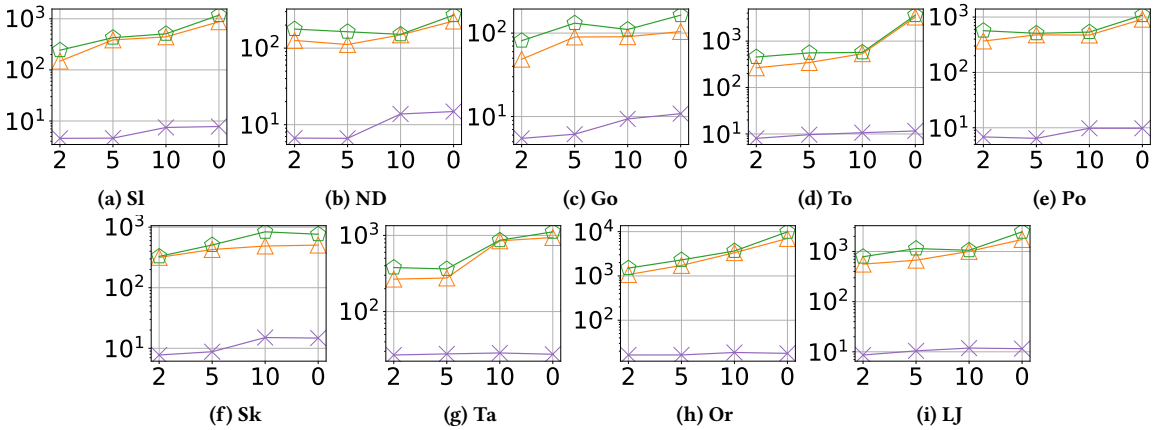


Figure 6: Average update time ($\times 10^{-6}$ second) vs. η

Query time vs. error. Recall that the algorithms run L steps recursively to approximate $\boldsymbol{\pi}$. Next, we measure the relative errors and the running time consumed over these L iterations. The ground truth is gained by running the deterministic algorithm for L' steps until $\hat{\boldsymbol{\pi}}_{L'}(u) - \hat{\boldsymbol{\pi}}_{L'-1}(u) < 10^{-9}$ for all $u \in V$. The average relative errors (ARE's) for all vertices vs. the running time on PageRank are reported in Figure 2. We observe that all algorithms have achieved the target relative error bound, which is 10% as $c = 0.1$ within L steps. Specifically, on ND, Go, Po, and LJ, the ARE's are below 3%. Since ε is set to be different values, recall that $\varepsilon = O(\frac{\delta}{L})$ in our AGP-Static++ and AGP-Dynamic, while $\varepsilon = O(\frac{\delta}{L^2})$ in AGP-Static. We observe that our methods incur slightly higher errors than AGP-Static after L steps. This is because with a smaller ε, AGP-Static performs more deterministic propagation (with higher running time) and hence potentially introduces less error. Nonetheless, when considering the query time, our algorithms reach low relative errors significantly faster, except on Ta. In particular, as shown in Figure 2 (i), on the largest graph, LJ, AGP-Static++ achieved an ARE of 3.085% in 187 seconds, whereas AGP-Static required 565 seconds to achieve 3.537% ARE. However, on Ta, its small average degree ($\bar{d} = 3.90$ as shown in Table 2) limits the efficiency gain from the randomized propagations for queries, resulting in comparable performance across all three algorithms.

7.2 Updating Efficiency

Competitors. We compare our *AGP-Dynamic* and *AGP-Static++* with *AGP-Static*. *AGP-DPSS* uses the same update method as *AGP-Static++* but with a larger hidden constant, and is omitted here.

Update Generation. We randomly generate a sequence of edge insertions and deletions for each dataset. We examine the impact of the insertion-to-deletion ratio η by assigning the probabilities of an insertion and a deletion as $\frac{\eta}{1+\eta}$ and $\frac{1}{1+\eta}$, respectively. To delete an edge, we uniformly sample an existing edge and delete it. To insert an edge, we adopt three strategies. **Random-Random (RR)**: A non-existent edge is inserted by randomly selecting two unconnected vertices. **Degree-Random (DR)**: A vertex u is selected with probability $\frac{d_u}{2m}$, where d_u is the degree of u and m is the current number of edges. Then, a vertex v not yet connected to u is chosen uniformly at random. **Degree-Degree (DD)**: Vertex u is selected as

in DR. Then, vertex v is selected from the vertices not yet connected to u, with probability proportional to $\frac{d_v}{2m}$.

By default, we set the insertion-to-deletion ratio to 10:1 ($\eta = 10$) and adopt the **DR** strategy, reflecting the common real-world scenario where insertions are more frequent than deletions, and users tend to follow high-degree nodes (celebrity) in social networks. We create $2m$ updates for each dataset and each η and update pattern, where m is the number of edges of the input graph.

Results. As shown in Figure 4, our *AGP-Dynamic* outperforms *AGP-Static++* and *AGP-Static* across all datasets in update efficiency. Compared to *AGP-Static++* and *AGP-Static*, the speedup ranges from 8× (on Go) to 177× (on Or). This trend roughly matches the average degree of each dataset as shown in Table 2, which is consistent with the theoretical improvement from the $O(d_u)$ update time of existing solutions to our amortized $O(1)$ update time. The results for *AGP-Static++* and *AGP-Static* indicate that updates are more costly on graphs with a larger average degree, which is expected since a change in d_u can incur an $O(d_u)$ cost. Additionally, *AGP-Static* maintains a sorted linked list which pays an extra $O(\log n)$ update time, hence is slower.

Impact of Update Pattern. As further shown in Figure 6, overall, the update times for *AGP-Static++* and *AGP-Static* increase as updates become more skewed (from RR to DR to DD). This is expected, as updates on higher-degree vertices incur a larger cost (i.e., $O(d_u)$). However, this trend does not always hold for *AGP-Dynamic*. When the degree of a vertex increases, the threshold $2 \cdot \tilde{d}_u$ (or $\frac{1}{2} \cdot \tilde{d}_u$) also increases, making it harder to trigger the $O(d_u)$ cost in *AGP-Dynamic*. Notably, *AGP-Dynamic* outperforms the baseline methods on all datasets for all update patterns by up to 207× (DD on Or).

Impact of Insertion-to-Deletion Ratio. Similarly, as shown in Figure 6, when the ratio of insertion increases, the update times for *AGP-Static++* and *AGP-Static* increase ($\eta = 0$ means no deletions), while our *AGP-Dynamic* has a more stable trend and outperforms *AGP-Static* by up to 553× on Or when $\eta = 0$. Additionally, on larger datasets, our *AGP-Static* performs more stably. This is because there are more updates ($2m$) on larger datasets, which justify our amortized cost.

Impact of Update-to-Query Frequency Ratio. From the above experiments, we can see that *AGP-Dynamic* is less efficient than *AGP-Static++*in query time while it outperforms *AGP-Static++* in update cost. To further study the trade off, we compare the update and query times of *AGP-Dynamic* and *AGP-Static++* in Table 3. The results show that *AGP-Dynamic* is at most 36% slower than *AGP-Static++* (i.e., 1.36× the time). We further examine the update-to-query frequency ratio under which *AGP-Dynamic* outperforms *AGP-Static++*. This ratio depends on the average degree $\bar{d}$ of the dataset. For instance, in Ta with $\bar{d} = 3.9$, a relatively high update-to-query ratio is required for *AGP-Dynamic* to gain an advantage. In contrast, in Or with $\bar{d} = 76.22$, only 206 updates per query are sufficient for *AGP-Dynamic* to become more efficient. On large graphs such as LJ, which contains over 1 billion edges, *AGP-Dynamic* outperforms *AGP-Static++* with as few as 754 updates per query. Since updates are usually much more frequent than queries in practice, *AGP-Dynamic* remains highly competitive.

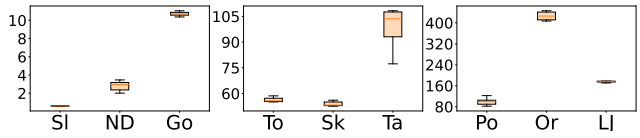


Figure 7: Query time (sec) on random a and b of *AGP-Static++*

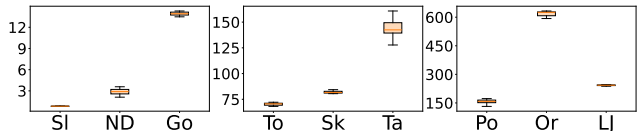


Figure 8: Query time (sec) on random a and b of *AGP-Dynamic*

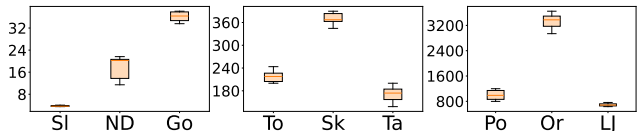
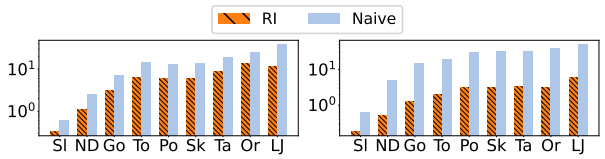


Figure 9: Query time (sec) on random a and b of *AGP-Static*



(a) Random x **(b) PageRank**

Figure 10: Average initialization time ($\times 10^{-3}$ second)

7.3 Arbitrary a and b

For each dataset, we execute 100 queries where a and b are drawn from the uniform distribution over $[0, 1]$, subject to the constraint $a+b \geq 1$. We set $w_i = \alpha(1-\alpha)^i$ with $\alpha = 0.2$, and $\mathbf{x}$ to a one hot vector generated uniformly at random. The results for *AGP-Dynamic* are shown in Figure 7. As expected, the query times vary across different (a, b) combinations, since according to Theorem 1, a and b impact the output size and, consequently, the query complexity. Nevertheless, as shown in Figure 7, *AGP-Dynamic* consistently handles all queries within the same order of magnitude across datasets, with standard deviations ranging from 1.6% to 33% of the average query time (the highest variation is observed on ND). The performance of *AGP-Static++* and *AGP-Static* exhibits similar trends but with slower response times, which are shown in Figure 8 and Figure 9, respectively. *AGP-DPSS* cannot support a and b given on the fly without rebuilding the data structures, and hence it is omitted.

7.4 Initialization Efficiency

Competitors. We compare our randomized algorithm (denoted as *RI*) with the naive deterministic initialization approach.

Generate x. We generate $\mathbf{x}$ in the form of P and S in three steps. First, generate the group number $|P|$ in $[1, \frac{1}{\varepsilon}]$ uniformly at random. Second, for each group, generate group size $|P_i|$ following uniform distribution with $\sum_{i=1}^{|P|} |P_i| = n$. Last, for each group, generate a

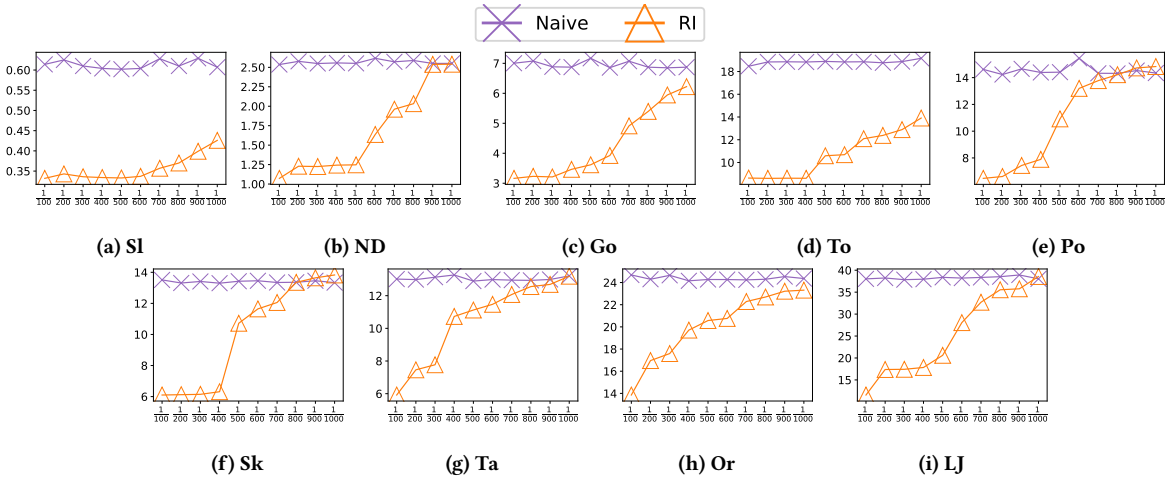


Figure 11: Average initialization time ($\times 10^{-3}$ second) vs. δ

value s_i following uniform distribution satisfying $\sum_{i=1}^{|P|} s_i = 1$, and then set the value of each group $S_i = \frac{s_i}{|P_i|}$.

For each dataset, we generate 100 random vectors $\mathbf{x}$ and set $\delta = \frac{1}{100}$. As shown in Figure 10a, *RI* consistently outperforms the *Naive* approach across all datasets, achieving up to a 3.29× speedup on LJ. We also evaluate our algorithm in the context of the PageRank application, where $\mathbf{x} = \left[\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right]^T$. In this setting, our randomized initialization method exhibits even greater advantages, yielding at least a 2× speedup on Sl and up to an 11× improvement on Or, without compromising the approximation quality. These results highlight the practical benefits of our randomized initialization strategy.

As shown in Figure 11, when δ decreases, the performance of *RI* degrades as expected, since a tighter error bound necessitates more direct assignments rather than sampling. Nevertheless, even with $\delta = \frac{1}{1000}$, *RI* still achieves up to a 1.42× speedup.

8 CONCLUSION

We revisited the Approximate Graph Propagation problem, which generalizes different graph propagation applications through a unified formulation. For the static case of the problem without graph updates, we derived a tighter query time complexity bound, improving over the state of the art by a factor up to $O(\log^2 n)$ and resulting in a 10× increase in query efficiency empirically. For the dynamic case with graph updates, our proposed *AGP-Dynamic* algorithm supports flexible query parameters given on-the-fly and attains an amortized $O(1)$ update time, accelerating update efficiency by up to 177× empirically.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Aleksandar Bojchevski, Johannes Gasteiger, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. 2020. Scaling graph neural networks with approximate pagerank. In *KDD*. 2464–2473.

[3] Karl Bringmann and Tobias Friedrich. 2013. Exact and efficient generation of geometric random variates and random graphs. In *International Colloquium on Automata, Languages, and Programming*. 267–278.

[4] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* 15, 3 (2024), 1–45.

[5] Fan Chung. 2007. The heat kernel as the PageRank of a graph. *Proceedings of the National Academy of Sciences* 104, 50 (2007), 19735–19740.

[6] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. 2024. From local to global: A graph RAG approach to query-focused summarization. *arXiv preprint arXiv:2404.16130* (2024).

[7] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A survey on RAG meeting LLMs: Towards retrieval-augmented large language models. In *KDD*. 6491–6501.

[8] Martín Farach-Colton and Meng-Tsung Tsai. 2015. Exact sublinear binomial sampling. *Algorithmica* 73 (2015), 637–651.

[9] Kurt C Foster, Stephen Q Muth, John J Potterat, and Richard B Rothenberg. 2001. A faster katz status score algorithm. *Computational & Mathematical Organization Theory* 7 (2001), 275–285.

[10] Junhao Gan, Seeun William Umboh, Hanzhi Wang, Anthony Wirth, and Zhuo Zhang. 2024. Optimal Dynamic Parameterized Subset Sampling. *Proceedings of the ACM on Management of Data* 2, 5 (2024), 1–26.

[11] Jaime Israel García-García, Nicolás Alonso Fernández Coronado, Elizabeth H Arredondo, and Isaac Alejandro Imilpán Rivera. 2022. The binomial distribution: Historical origin and evolution of its problem situations. *Mathematics* 10, 15 (2022), 2680.

[12] Johannes Gasteiger, Aleksandar Bojchevski, and Stephan Günnemann. 2019. Predict then propagate: Graph neural networks meet personalized pagerank. In *ICLR*.

[13] Johannes Gasteiger, Stefan Weißenberger, and Stephan Günnemann. 2019. Diffusion improves graph learning. In *NeurIPS*. 13366–13378.

[14] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[15] Bernal Jiménez Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. 2024. HippoRAG: Neurobiologically inspired long-term memory for large language models. In *NeurIPS*.

[16] Zoltán Gyöngyi, Hector Garcia-Molina, and Jan Pedersen. 2004. Combating web spam with TrustRank. In *VLDB*. VLDB Endowment, 576–587.

[17] Yiqian Huang, Shiqi Zhang, and Xiaokui Xiao. 2025. KET-RAG: A Cost-Efficient Multi-Granular Indexing Framework for Graph-RAG. *arXiv preprint arXiv:2502.09304* (2025).

[18] Jinhong Jung, Namyong Park, Sael Lee, and U Kang. 2017. BePI: Fast and memory-efficient method for billion-scale random walk with restart. In *SIGMOD*. 789–804.

[19] Voratas Kachitvichyanukul and Bruce W Schmeiser. 1988. Binomial random variate generation. *Commun. ACM* 31, 2 (1988), 216–222.

[20] Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences* 103 (2023), 102274.

[21] Michael E Kuhl. 2017. History of random variate generation. In *WSC*. IEEE, 231–242.

[22] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

[23] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *NeurIPS*. 9459–9474.

[24] Yinheng Li, Shaofei Wang, Han Ding, and Hang Chen. 2023. Large language models in finance: A survey. In *ACM International Conference on AI in Finance*. 374–382.

[25] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. DeepSeek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[26] Peter Lofgren and Ashish Goel. 2013. Personalized PageRank to a target node. *arXiv preprint arXiv:1304.4658* (2013).

[27] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank Citation Ranking: Bringing Order to the Web. In *WWW*. https://api.semanticscholar.org/CorpusID:1508503

[28] Sheldon M Ross. 2020. *A first course in probability*. Pearson.

[29] Shamane Siriwardhana, Rivindu Weerasekera, Elliott Wen, Tharindu Kaluarachchi, Rajib Rana, and Suranga Nanayakkara. 2023. Improving the domain adaptation of retrieval augmented generation (RAG) models for open domain question answering. *Transactions of the Association for Computational Linguistics* 11 (2023), 1–17.

[30] Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. 2023. Large language models in medicine. *Nature Medicine* 29, 8 (2023), 1930–1940.

[31] Lloyd N. Trefethen and David Bau. 2022. *Numerical inear algebra*. SIAM.

[32] Hanzhi Wang, Mingguo He, Zhewei Wei, Sibo Wang, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. 2021. Approximate graph propagation. In *KDD*. 1686–1696.

[33] Yu Wang, Nedim Lipka, Ryan A Rossi, Alexa Siu, Ruiyi Zhang, and Tyler Derr. 2024. Knowledge graph prompting for multi-document question answering. In *AAAI*. 19206–19214.

[34] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying graph convolutional networks. In *ICML*. 6861–6871.

[35] Junde Wu, Jiayuan Zhu, Yunli Qi, Jingkun Chen, Min Xu, Filippo Menolascina, and Vicente Grau. 2024. Medical graph RAG: Towards safe medical large language model via graph retrieval-augmented generation. *arXiv preprint arXiv:2408.04187* (2024).

[36] Zhuowei Zhao. 2025. Source code:https://github.com/alvinzhaowei/AGP-dynamic. https://github.com/alvinzhaowei/AGP-dynamic