Efficient and scalable techniques for pagerank-based graph analytics

Yang, Renchi

2020

Yang, R. (2020). Efficient and scalable techniques for pagerank-based graph analytics.
Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/145185>

<https://doi.org/10.32657/10356/145185>

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0
International License (CC BY-NC 4.0).

Downloaded on 26 Apr 2023 16:25:09 SGT

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

**EFFICIENT AND SCALABLE
TECHNIQUES FOR PAGERANK-BASED
GRAPH ANALYTICS**

YANG Renchi

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2020

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

July 21, 2020

.....

Date

.....

Yang Renchi

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

July 21, 2020

.....

Date

.....

Sourav Saha Bhowmick

Authorship Attribution Statement

Please select one of the following; *delete as appropriate:

*(B) This thesis contains material from three paper(s) published in the following peer-reviewed journal(s) / from papers accepted at conferences in which I am listed as an author.

Chapter 4 is published as **Renchi Yang**, Xiaokui Xiao, Zhewei Wei, Sourav Bhowmick, Jun Zhao, and Rong-Hua Li, "Efficient Estimation of Heat Kernel PageRank for Local Clustering", Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 1339-1356, 2019.

The contributions of the co-authors are as follows:

- Associate Professor Xiaokui XIAO provided the initial project direction and the basic idea of the work.
- I prepared the manuscript drafts. The manuscript was revised by Associate Professor Xiaokui XIAO, Associate Professor Sourav Bhowmick, Assistant Professor Jun ZHAO.
- I co-designed the algorithm with Associate Professor Xiaokui XIAO, implemented all the algorithms and performed all the experiments at the School of Computer Science and Engineering.
- Professor Zhewei WEI and Assistant Professor Jun ZHAO assisted in the proofs and algorithmic design.

Chapter 5 is published as **Renchi Yang**, Jieming Shi, Xiaokui Xiao, Yin Yang, Sourav S. Bhowmick, "Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank", Proceedings of the VLDB Endowment (PVLDB), pp. 670-683, 2020.

The contributions of the co-authors are as follows:

- Associate Professor Xiaokui XIAO provided the initial project direction and the basic idea of the work.

- I wrote the drafts of the manuscript. The manuscript was revised together with Assistant Professor Jieming SHI, Associate Professor Xiaokui XIAO, Associate Professor Yin YANG and Associate Professor Sourav Bhowmick.
- I implemented our algorithms, collected all the datasets and codes of all competitors, and conducted all experimental evaluation.
- Assistant Professor Jieming SHI and Associate Professor Xiaokui XIAO assisted in the proofs and algorithmic design.

July 21, 2020

.....

Date

.....

Yang Renchi

Abstract

Graphs are ubiquitous today and are a fundamental data structure to represent objects and their relations in various domains, *e.g.*, social science, citation analysis, weblink analysis, and biological informatics. PageRank-based techniques such as *personalized PageRank*, *heat kernel PageRank*, *TrustRank* have been well studied and shown great efficacy in graph processing tasks including web ranking, recommender system, graph clustering and combating web spam. In this thesis, we investigate three important problems in graph processing by exploiting PageRank-based techniques, namely, local graph clustering, homogeneous network embedding and attributed network embedding. These three problems are not only interesting in themselves when the graph size becomes large, but also find numerous applications in both academia and industry.

First, we study the local graph clustering on undirected graphs. Given an undirected graph G and a *seed* node s , the *local clustering* problem aims to identify a high-quality cluster containing s in time roughly proportional to the size of the cluster, regardless of the size of G . Recently, *heat kernel PageRank* (HKPR) is applied to this problem and found to be more efficient compared with prior methods. However, existing solutions for computing HKPR either are prohibitively expensive or provide unsatisfactory error approximation on HKPR values, rendering them impractical especially on billion-edge graphs. Thus, we present TEA and TEA+, which utilize deterministic graph traversal to produce a rough estimation of exact HKPR vector, and then exploit Monte-Carlo random walks to refine the results in an optimized and non-trivial way. In particular, TEA+ offers practical efficiency and effectiveness due to non-trivial optimizations.

Second, we investigate the *homogeneous network embedding* (HNE) problem. Given an input graph G and a node $v \in G$, HNE aims to map the graph structure in the vicinity of v to a fixed-dimensional feature vector. Existing approaches to HNE are either immensely expensive, and, thus, are limited to small graphs or fail to fully capture

the local graph structure, leading to limited effectiveness of the extracted feature vectors. Meanwhile, in recent years there have been rapid advancements in scalable algorithms for computing approximate *personalized PageRank* (PPR), which captures rich graph topological information. However, PPR was designed for a very different purpose, *i.e.*, ranking nodes in G based on their relative importance from a source node’s perspective. In contrast, HNE aims to build node embeddings considering the whole graph. Consequently, node embeddings derived directly from PPR are of suboptimal utility. Motivated by this, we propose Node-Reweighted PageRank (NRP), a novel solution that transforms PPR values into effective node embeddings, by iteratively solving an optimization problem.

Finally, we consider the *attributed network embedding* (ANE) problem. Given a graph G where each node is associated with a set of attributes, ANE maps each node $v \in G$ to a compact vector X_v , which can be used in downstream machine learning tasks. Existing solutions largely fail on such graphs, leading to prohibitive costs, low-quality embeddings, or both. We propose PANE, an effective and scalable approach to ANE computation for massive graphs. PANE obtains high scalability and effectiveness through three main algorithmic designs. First, it formulates the learning objective based on a novel *node-attribute PageRank* model for attributed networks. The resulting optimization task is still challenging on large graphs. Second, PANE includes a highly efficient solver for the above optimization problem, whose key module is a carefully designed initialization of the embeddings, which drastically reduces the number of iterations required to converge. Finally, PANE utilizes multi-core CPUs through non-trivial parallelization of the above solver, which achieves scalability while retaining the high quality of the resulting embeddings.

Acknowledgments

I would like to express my deepest gratitude to my supervisors Dr. Xiaokui Xiao and Dr. Sourav Saha Bhowmick for their patience, enthusiasm, knowledgeable insights and continuous support throughout my Ph.D. study. It has been a wonderful experience studying and working under the supervision of Dr. Xiao and Dr. Bhowmick in the past years, during which I have learned not only cutting-edge knowledge on database and data management, but also an inspiring and down-to-earth working attitude.

I would also like to thank all my collaborators and colleagues for their encouragement, friendships and insightful discussions. They include but not limited to Dr. Sibor Wang at Chinese University of Hong Kong, Dr. Zhewei Wei at Renmin University of China, Dr. Yin Yang at Hamad Bin Khalifa University, Dr. Jun Zhao at Nanyang Technological University, Dr. Jieming Shi at Hong Kong Polytechnic University, Dr. Xiaoqi Yan at Singapore Technologies Engineering, Dr. Jigang Liu at Nanyang Technological University, Dr. Dezhong Yao at Huazhong University of Science and Technology, Dr. Keke Huang and Dr. Xi Lin.

Last but not least, great appreciation to my family for their constant encouragement, understanding, support and love. I am sure that I won't achieve what I have achieved today without them.

Contents

Abstract	i
Acknowledgments	iii
List of Figures	viii
List of Tables	x
1. Introduction	1
1.1 Background	1
1.2 Problems and Motivation	2
1.3 Approaches and Contributions	7
1.4 Thesis Organization	9
2. Preliminaries	12
2.1 Notations and Terminology	12
2.2 PageRank and Personalized PageRank	13
2.3 Heat Kernel PageRank	15
2.4 Local Graph Clustering	15
2.5 Network Embedding	17
3. Literature Review	19
3.1 Local Graph Clustering	19
3.1.1 ClusterHKPR	19
3.1.2 HK-Relax	20
3.1.3 Other Methods for Local Clustering	21

3.1.4	Methods for Personalized PageRank	21
3.2	Homogeneous Network Embedding	23
3.2.1	Random Walk-based Methods	23
3.2.2	Deep Learning-based Methods	24
3.2.3	Matrix Factorization-based Methods	24
3.2.4	Other Methods	26
3.3	Attributed Network Embedding	26
3.3.1	Factorization-based methods	26
3.3.2	Auto-encoder-based methods	27
3.3.3	Other methods	28
4.	Local Graph Clustering via Heat Kernel PageRank	29
4.1	Problem Formulation	30
4.1.1	Heat-Kernel-PageRank-based Local Clustering	30
4.1.2	Solution Objective	31
4.2	The TEA Algorithm	33
4.2.1	HK-Push	34
4.2.2	Algorithm	38
4.2.3	Analysis	42
4.3	The TEA+ Algorithm	45
4.3.1	The Case without Random Walks	46
4.3.2	The Case with Random Walks	49
4.3.3	Details of TEA+	50
4.3.4	Example for TEA+	51
4.3.5	Analysis	54
4.4	Experiments	58
4.4.1	Experimental Setup	58
4.4.2	Tuning Parameter c for TEA+	59
4.4.3	Comparison between TEA and TEA+	60
4.4.4	Comparisons with Competitors	62
4.4.5	Ranking Accuracy of Normalized HKPR	66
4.4.6	Clusters Produced vs. Ground-truth	68

4.4.7	Sensitivity Analysis to the Subgraph Characteristics	68
4.4.8	Effects of Heat Constant t	69
4.5	Summary	70
5.	Homogeneous Network Embedding via Node-Reweighted PageRank	71
5.1	Problem Formulation	72
5.1.1	Homogeneous Network Embedding	72
5.1.2	Solution Objective	73
5.2	The Baseline Algorithm	74
5.3	Proposed NRP Algorithm	79
5.3.1	Choice of Node Weights	80
5.3.2	Learning Node Weights	81
5.3.3	Accelerating Weight Updates	83
5.3.4	Updating Forward Weights	87
5.3.5	Complete NRP Algorithm and Analysis	91
5.4	Experiments	92
5.4.1	Experimental Settings	92
5.4.2	Link Prediction	94
5.4.3	Graph Reconstruction	97
5.4.4	Node Classification	98
5.4.5	Efficiency	99
5.4.6	Parameter Analysis	100
5.4.7	Additional Experiments	101
5.5	Summary	103
6.	Attributed Network Embedding via Node-Attribute PageRank	105
6.1	Problem Formulation	106
6.1.1	Attributed Network Embedding	106
6.1.2	Node-Attribute PageRank	108
6.1.3	Solution Objective	110
6.2	The PANE Algorithm	111
6.2.1	Forward and Backward Node-Attribute PageRank Approximation	112

6.2.2	Joint Factorization of Node-Attribute PageRank Matrices	115
6.2.3	Complexity Analysis	118
6.3	Parallelization of PANE	119
6.3.1	Parallel Forward and Backward Node-Attribute PageRank Approx- imation	120
6.3.2	Parallel Joint Factorization of Node-Attribute PageRank Matrices	123
6.3.3	Complexity Analysis	126
6.4	Experiments	127
6.4.1	Experiments Setup	127
6.4.2	Attribute Inference	130
6.4.3	Link Prediction	131
6.4.4	Node Classification	133
6.4.5	Efficiency and Scalability	134
6.4.6	Paramter Analysis	136
6.4.7	Effectiveness Evaluation of Greedylnit	136
6.5	Summary	138
7.	Conclusion and Future Work	139
7.1	Conclusions	139
7.2	Future Directions	140
7.2.1	Local Clustering on Attributed Graphs.	141
7.2.2	Attributed Heterogeneous Network Embedding	142
7.2.3	Dynamic Homogeneous/Attributed Network Embedding	143
	Publications	145
	References	147

List of Figures

1.1	An example graph G	5
2.1	An example social graph G for local graph clustering.	16
2.2	An example social graph G for HNE.	17
4.1	A graph G' for illustrating TEA+.	52
4.2	Running time of TEA+ vs c (best viewed in color).	60
4.3	Running time vs ϵ_r	60
4.4	Running time vs conductance for local clustering queries (best viewed in color).	61
4.5	Memory cost vs. conductance.	63
4.6	Running time vs. NDCG for computing normalized HKPR (best viewed in color).	63
4.7	Effect of subgraph densities (best viewed in color).	64
4.8	Effect of heat constant t on $DBLP$ (best viewed in color).	66
4.9	Effect of heat constant t on PLC (best viewed in color).	67
5.1	Illustration of Example 5.1 for the Baseline algorithm.	79
5.2	Illustration for Example 5.2	85

5.3	Link prediction results vs. embedding dimensionality k (best viewed in color).	95
5.4	Graph reconstruction results vs. K (best viewed in color).	96
5.5	Node classification results (best viewed in color).	97
5.6	Running time vs. embedding dimensionality k (best viewed in color).	99
5.7	Link prediction results with varying parameters (best viewed in color).	101
5.8	Link prediction performance on dynamic graphs (best viewed in color).	102
5.9	Scalability tests.	102
5.10	Running time with varying parameters (best viewed in color).	102
6.1	Extended graph $\mathcal{G}$	108
6.2	Node classification results (best viewed in color).	129
6.3	Running time (best viewed in color).	133
6.4	Running time with varying parameters.	134
6.5	Attribute inference results with varying parameters (best viewed in color).	135
6.6	Link prediction results with varying parameters (best viewed in color).	135
6.7	Effectiveness of GreedyInIt in Link Prediction.	137
6.8	Effectiveness of GreedyInIt in Attribute Inference.	137

List of Tables

1.1	PPR for $\mathbf{v}_2$ and $\mathbf{v}_9$ in Fig. 1.1 ($\alpha = 0.15$).	5
2.1	Frequently used notations.	13
3.1	Theoretical guarantees of our solutions against that of the state-of-the-art solutions.	20
4.1	Frequently used notations.	30
4.2	An example for TEA+	49
4.3	The reserves and residues after the first round of push operations.	52
4.4	The reserves and residues after the second round of push operations.	53
4.5	The reserves and residues after the third round of push operations.	53
4.6	Statistics of graph datasets.	59
4.7	The result of evaluating all algorithms on finding real-world communities.	64
5.1	Frequently used notations.	73
5.2	Dataset statistics ($K = 10^3$, $M = 10^6$, $B = 10^9$).	93
5.3	Dataset statistics ($K = 10^3$, $M = 10^6$).	101
6.1	Frequently used notations.	107
6.2	Targets for $\mathbf{X}[v_i] \cdot \mathbf{Y}[r_j]^\top$	108
6.3	Datasets. ($K=10^3$, $M=10^6$)	127
6.4	Attribute inference performance.	127
6.5	Link prediction performance.	128

Chapter 1

Introduction

1.1 Background

Graphs are ubiquitous today and are a fundamental data structure to represent objects and their relations in various domains, *e.g.*, social science, citation analysis, weblink analysis, and biological informatics. Given a graph $G = (V, E)$, where V (resp. E) is the set of nodes (resp. edges) in G , a source node $s \in V$, a target node $t \in V$, and a probability α , the personalized PageRank [40, 68, 78] $\pi(s, t)$ of t with respect to s is defined as the probability that a random walk on G from s terminates at t , which can be understood as the importance of t from the perspective of s (see details in Section 2.2 of Chapter 2). PPR well captures the graph topology and finds numerous applications in graph processing tasks such as Twitter’s Who-To-Follow [65], LinkedIn’s connection recommendation [122], Pinterest’s Related Pins [96] and many others [55]. Besides, many variants of PPR are specially devised for different graph processing tasks to achieve improved performance, such as *heat kernel PageRank* for local graph clustering [36], *TrustRank* for combating web spam [66], and *Triangle Reinforced PageRank* for pairwise link prediction [106]. However, exploiting these PageRank-based techniques and making connections from them to addressing specific graph processing tasks remains challenging especially when graphs are sizable because the computation is immensely expensive. We observe that, in recent years, there have been rapid advancements [45, 47, 48, 64, 82, 124, 125, 147, 149, 173, 174] in scalable algorithms for computing approximate PPR, which enable us to design scalable PPR-based techniques for different graph processing tasks over massive graphs as well.

In this thesis, we investigate three popular problems in graph processing, which are local graph clustering (see Definition 2.2), homogeneous network embedding (see Definition 2.3), and attributed network embedding (see Definition 2.4). Though these problems are widely studied in both industry and academia, none of the existing methods to them scale to massive graphs on commodity hardware without compromising result utility. Towards this end, we address them by carefully devising scalable PageRank-based techniques. In what follows, this chapter will briefly introduce the problems we study and our proposed solutions. More details will be presented in later chapters respectively.

1.2 Problems and Motivation

Local Graph Clustering. Graph clustering is a fundamental problem that finds numerous applications, *e.g.*, community detection [44, 50, 93], image segmentation [43, 49], and protein grouping in biological networks [38, 140]. The problem has been studied extensively in the literature, and yet, clustering massive graphs remains a challenge in terms of computation efficiency. This motivates a series of algorithms [3, 4, 6, 21, 36, 42, 46, 54, 107, 128, 137, 190] for *local clustering*, which takes as input an undirected graph G and a *seed* node s , and identifies a cluster (*i.e.*, a set of nodes) containing s in time depending on the size of the cluster, regardless of the size of G .

Local clustering algorithms have the potential to underpin interactive exploration of massive graphs. Specifically, they can facilitate the exploration of a relatively small region of a large graph that is of interest to a user. For example, consider Bob, a budding entrepreneur, who is interested in exploring the local clusters of visionary entrepreneurs on *Twitter*. Particularly, he wishes to begin his exploration with the cluster associated with **Elon Musk** (*i.e.*, seed). Since Bob thinks that Elon Musk is an inspirational entrepreneur, he would like to explore if there are any other notable entrepreneurs (*e.g.*, **Kevin Rose**) in Elon’s local cluster (*e.g.*, followers, followees) and wishes to further explore the local neighborhoods of these entrepreneurs. In order to ensure a palatable and non-disruptive interactive experience, Bob needs an efficient local clustering framework that can return high-quality clusters within few seconds. *Which existing local clustering framework can Bob utilize for his exploration?*

Spielman and Teng [42, 128] are the first to study the local clustering problem, and they propose a random-walk-based algorithm, **Nibble**, that optimizes the *conductance* [10] of the cluster returned. Specifically, the conductance of cluster S is defined as $\Phi(S) = \frac{|\text{cut}(S)|}{\min\{\text{vol}(S), 2m - \text{vol}(S)\}}$, where $\text{vol}(S)$ is the sum of the degrees of all nodes in S , m is the number of edges in the graph G , and $|\text{cut}(C)|$ is number of edges with one endpoint in S and another not in S . Intuitively, if a cluster C has a small conductance, then the nodes in S are better connected to each other than to the nodes apart from S , in which case S should be considered a good cluster. This algorithm is subsequently improved in a series of work [3, 4, 21, 46, 54, 107, 128, 137, 190] that aims to either improve the efficiency of local clustering or reduce the conductance of the cluster returned.

The state-of-the-art solutions [21, 36] for local clustering are based on *heat kernel PageRank* [18], which is a measure of the proximity of nodes in graphs. Given a seed node s , these solutions first compute a vector $\hat{\rho}_s$ where each element $\hat{\rho}_s[v]$ approximates the HKPR value of a node v with respect to s (i.e., $\hat{\rho}_s[v]$ approximately measures the proximity of s to v). Then, they utilize $\hat{\rho}_s$ to derive a local cluster C containing s . It is shown that the quality of S depends on the accuracy of $\hat{\rho}_s$ [22, 36], in the sense that the conductance of S tends to decrease with the approximation error in $\hat{\rho}_s$. Therefore, existing HKPR-based solutions [21, 36] all focus on striking a good trade-off between time efficiency and the accuracy of $\hat{\rho}_s$. In particular, the current best solution HK-Relax [36] ensures that (i) $\frac{1}{d(v)} |\hat{\rho}_s[v] - \rho_s[v]| < \epsilon_a$ for any node v , where ϵ_a is a given threshold, $d(v)$ is the degree of node v and $\rho_s[v]$ is the exact HKPR value of node v with respect to s , and (ii) $\hat{\rho}_s$ is computed in $O\left(\frac{te^t \log(1/\epsilon_a)}{\epsilon_a}\right)$ time, where t is constant (referred to as the *heat constant*) used in the definition of HKPR.

The time complexity of HK-Relax has a large factor e^t , where t (i.e., the heat constant) could be as large as a few dozens [21, 36, 41]. Consequently, it can be inefficient for several applications. For instance, reconsider Bob's endeavor to explore the local clusters of Elon Musk and Kevin Rose. HK-Relax consumes around 15s and 48s, respectively, to compute their local clusters. Such performance is disruptive for any interactive graph exploration. In addition, HK-Relax provides an accuracy guarantee on each $\frac{\hat{\rho}_s[v]}{d(v)}$ in terms of its *absolute error*, but as we discuss in Chapter 4.1.2, this guarantee is less than ideal for accurate local clustering. Specifically, HKPR-based local clustering requires

ranking each node v in descending order of $\frac{\hat{\rho}_s[v]}{d(v)}$, which we refer to as v 's *normalized HKPR*. To optimize this accuracy of this ranking, we observe that it is more effective to minimize the *relative errors* of normalized HKPR values than their absolute errors. To explain, we note that the normalized HKPR varies significantly from nodes to nodes. For the aforementioned ranking, nodes with large normalized HKPR could tolerate more absolute errors than nodes with small normalized HKPR, and hence, imposing the same absolute error guarantees on all nodes tend to produce sub-optimal results.

Homogeneous Network Embedding. Given a graph $G = (V, E)$ with n nodes, a *network embedding* maps each node $v \in G$ to a compact feature vector in $\mathbb{R}^k$ ($k \ll n$), such that the embedding vector captures the local graph features surrounding v . These embedding vectors are then used as inputs in downstream machine learning operations [133, 144, 158]. In this thesis, we study *homogeneous network embeddings* (HNEs), which reflect the topology of G rather than labels associated with nodes or edges. HNE methods have been commonly applied to a plethora of graph mining tasks, including node classification [116], link prediction [7], and graph reconstruction [108].

HNEs can be either *learned* from data or *constructed* from graph features. Learning-based approaches tend to be costly and difficult to scale to larger graphs. For instance, a classic method [110] is to learn node embeddings from a large number of random walk simulations. The larger the number of random walks, the better the quality of the learned embeddings. However, performing numerous random walks for each node in a large graph can be prohibitively expensive. Similarly, HNEs can be learned directly from the local graph structure through a deep neural network [142], which is costly to train and hard to scale to large graphs. In contrast, constructing HNEs from hand-crafted graph features (*e.g.*, obtained by factorizing a proximity matrix [183]) can be considerably faster. Unfortunately, as we review in Chapter 3, existing work either uses expensive features that are unscalable, or fast-and-rough ones that capture limited topological information, leading to less effective embeddings.

Is it possible to construct node embeddings efficiently and scalably that capture similarly rich topological information as in learning-based methods? This thesis explores a promising idea: constructing HNEs through approximate *personalized PageRank* [79]. Specifically, given a pair of nodes $u, v \in G$, the PPR value $\pi(u, v)$ of v with respect to u

Figure 1.1: An example graph G .

Table 1.1: PPR for v_2 and v_9 in Fig. 1.1 ($\alpha = 0.15$).

v_i	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$\pi(v_2, v_i)$	0.13	0.304	0.13	0.154	0.155	0.059	0.035	0.023	0.01
$\pi(v_9, v_i)$	0.017	0.03	0.017	0.03	0.054	0.088	0.17	0.313	0.283

is the probability that a random walk from u terminates at v . Therefore, PPR values can be viewed as a concise summary of an infinite number of random walk simulations, which captures extensive topological information as in random-walk-based learning methods. In fact, PPR has been applied in graph neural networks to achieve superior results compared to traditional graph convolutional networks [87]. Computing PPR values used to be rather expensive. Fortunately, in recent years the database community have developed fast and scalable approximate PPR algorithms [148, 150, 155]. This combination of rich information and high scalability renders PPR a particularly appealing choice for HNE construction.

Building effective node embeddings from PPR values, however, turns out to be challenging. The root cause is that PPR was designed to serve a very different purpose, *i.e.*, ranking nodes in G based on their relative importance from a source node’s perspective. In essence, PPR is a *local* measure. On the other hand, the goal of HNE is to concisely summarize nodes from a global viewpoint. To illustrate this critical difference, consider the example in Figure 1.1, which contains 9 nodes v_1 - v_9 . Observe that between nodes v_2 and v_4 , there are three different paths connecting the two, through v_1 , v_3 and v_5 , respectively. In contrast, there is only one path connecting nodes v_9 and v_7 . Intuitively, if we were to predict a new edge in the graph, it is more likely to be (v_2, v_4) than (v_9, v_7) . However, as shown in Table 1.1, in terms of PPR values, we have $\pi(v_2, v_4) < \pi(v_9, v_7)$, which contradicts with the above intuition. This shows that PPR by itself is *not* an ideal proximity measure, at least for the task of link prediction, and an effective HNE scheme must overcome this deficiency.

Attributed Network Embedding. Network embedding is a fundamental task for graph analytics, which has attracted much attention from both academia (*e.g.*, [60, 111, 129]) and industry (*e.g.*, [90, 189]). In a nutshell, given an input graph G , network embedding converts each node $v \in G$ to a compact, fixed-length vector X_v , which captures the topological features of the graph around node v . In practice, however, graph data often comes with *attributes* associated to nodes. While we could treat graph topology and attributes as separate features, doing so loses the important information of *node-attribute affinity* [101], *i.e.*, attributes that can be reached by a node through one or more hops along the edges in G . For instance, consider a graph containing companies and board members. An important type of insights that can be gained from such a network is that one company (*e.g.*, Tesla Motors) can reach attributes of another related company (*e.g.*, SpaceX) connected via a common board member (Elon Musk). To incorporate such information, *attributed network embedding* maps both topological and attribute information surrounding a node to an embedding vector, which facilitates accurate predictions, either through the embeddings themselves or in downstream machine learning tasks.

Effective ANE computation is a highly challenging task, especially for massive graphs, *e.g.*, with millions of nodes and billions of edges. In particular, each node $v \in G$ could be associated with a large number of attributes, which corresponds to a high dimensional space; further, each attribute of v could influence not only v 's own embedding, but also those of v 's neighbors, neighbors' neighbors, and far-reaching connections via multiple hops along the edges in G . Existing ANE solutions are immensely expensive and largely fail on massive graphs. Specifically, as reviewed in Chapter 3, one class of previous methods *e.g.*, [75, 161, 164, 177], explicitly construct and factorize an $n \times n$ matrix, where n is the number of nodes in G . For a graph with 50 million nodes, storing such a matrix of double-precision values would take over 20 petabytes of memory, which is clearly infeasible. Another category of methods, *e.g.*, [51, 97, 109, 181], employ deep neural networks to extract higher-level features from nodes' connections and attributes. For a large dataset, training such a neural network incurs vast computational costs; further, the training process is usually done on GPUs with limited graphics memory, *e.g.*, 32GB on Nvidia's flagship Tesla V100 cards. Thus, for massive graphs, currently the only option is to compute ANE with a large cluster, *e.g.*, [189], which is financially costly and out of the reach of most researchers.

In addition, to our knowledge, all existing ANE solutions are designed for undirected graphs, and it is unclear how to incorporate edge direction information (*e.g.*, asymmetric transitivity [184]) into their resulting embeddings. In practice, many graphs are directed (*e.g.*, one paper citing another), and existing methods yield suboptimal result quality on such graphs, as shown in our experiments. Can we compute effective ANE embeddings on a massive, attributed, directed graph on a single server?

1.3 Approaches and Contributions

Local Graph Clustering. Motivated by the deficiency of existing solutions, we present an in-depth study on HKPR-based local clustering, and make the following contributions. First, we formalize the problem of approximate HKPR computation with probabilistic relative error guarantees, and pinpoint why none of the existing techniques could provide an efficient solution to this problem.

Second, based on our problem formulation, we propose two new algorithms, **TEA** and **TEA+**, both of which (i) take as input a seed node s , two thresholds ϵ_r, δ , and a failure probability p_f , and (ii) return an approximate HKPR vector $\hat{\rho}_s$ where each element $\hat{\rho}_s[v]$ with $\frac{\rho_s[v]}{d(v)} > \delta$ has at most ϵ_r relative error with at least $1 - p_f$ probability (*i.e.*, all *significant* HKPR values are accurately approximated with high probability). The core of **TEA** is an adaptive method that combines deterministic graph traversal with random walks to estimate normalized HKPR in a cost-effective manner, while **TEA+** significantly improves over **TEA** in terms of practical efficiency by incorporating a number of non-trivial optimization techniques. Both algorithms have a time complexity of $O\left(\frac{t \log(n/p_f)}{\epsilon_r^2 \cdot \delta}\right)$, which eliminates the exponential term e^t in **HK-Relax**'s efficiency bound (see more details in Section 3.1 of Chapter 3).

Third, we experimentally evaluate them against the state of the art, using large datasets with up to 65 million nodes and 1.8 billion edges. Our results show that **TEA+** is up to an order of magnitude faster than competing methods when achieving the same clustering quality. In particular, it can compute the local clusters of Elon Musk and Kevin Rose within 1.3s and 6.1s, respectively, thereby facilitating interactive exploration.

Homogeneous Network Embedding. Motivated by the deficiency of PPR, we propose a novel *Node-Reweighted PageRank* (NRP) approach to HNE computation. The main idea in NRP is to augment PPR values with additional *node reweighting vectors* that address the deficiency of PPR described above. These node reweighting vectors are learned from data, *i.e.*, they are obtained by iteratively solving an optimization problem. As we show in the thesis, this learning process is efficient and scalable, whose cost is small compared to PPR computations. Additionally, instead of using off-the-shelf PPR algorithms which tend to be complex, NRP employs a simple, fast and easily parallelizable solution for approximate PPR computation that is specifically designed for the purpose of HNE. Overall, it takes $O(k(m + kn) \log n)$ time and $O(m + nk)$ space to construct NRP embeddings, where n , m are the number of nodes and edges in the input graph G , and k is the target size of the each embedding vector.

To evaluate the performance of NRP, we have conducted extensive experiments on 6 popular real datasets, and compared the NRP against 16 previous HNE solutions. The results demonstrate that NRP is clearly one of the fastest algorithms on all datasets. In particular, for a social network with 1.2 billion edges, NRP computation finishes within 4 hours on a single core of a commodity machine. Moreover, on large graphs the result quality of NRP is clearly superior to all previous scalable HNE methods by a large margin. Perhaps a bit surprisingly, even on smaller graphs, NRP consistently outperforms all previous methods in terms of result quality.

Attributed Network Embedding. We present PANE, a novel solution that significantly advances the state of the art in ANE computation. Specifically, as demonstrated in our experiments, the embeddings obtained by PANE simultaneously achieve the highest prediction accuracy compared to previous methods for 3 common graph analytics tasks: attribute inference, link prediction, and node classification, on common benchmark graph datasets. On the largest Microsoft Academic Knowledge Graph (MAG) dataset, PANE is the only viable solution on a single server, whose resulting embeddings lead to 0.88 average precision (AP) for attribute inference, 0.965 AP for link prediction, and 0.57 micro-F1 for node classification. PANE obtains such results using 10 CPU cores, 1TB memory, and 12 hours running time.

PANE achieves effective and scalable ANE computation through three main contributions: a well-thought-out problem formulation based on a novel random walk model (*i.e.*, personalized PageRank), a highly efficient solver, and non-trivial parallelization of the algorithm. Specifically, as presented in Chapter 6, PANE formulates the ANE task as an optimization problem with the objective of approximating normalized multi-hop *node-attribute PageRank* using node-attribute co-projections [101, 102], guided by a shifted pairwise mutual information (shifted PMI) metric that is inspired by natural language processing techniques. The node-attribute PageRank between a given node-attribute pair is defined via a random walk model specifically adapted to attributed networks. Further, we incorporate edge direction information by defining separate forward and backward node-attribute PageRanks and embeddings. Solving this optimization problem is still immensely expensive with off-the-shelf algorithms, as it involves the joint factorization of two $O(n \cdot d)$ -sized matrices, where n and d are the numbers of nodes and attributes in the input data, respectively. Thus, PANE includes a novel solver with a key module that seeds the optimizer through a highly effective greedy algorithm, which drastically reduces the number of iterations till convergence. Finally, we devise non-trivial parallelization of the PANE algorithm, in order to utilize modern multi-core CPUs without significantly compromising result utility. Extensive experiments, using 8 real datasets and comparing against 10 existing solutions, demonstrate that PANE consistently obtains high-utility embeddings which lead to superior prediction accuracy for attribute inference, link prediction and node classification, at a fraction of the cost compared to existing methods.

1.4 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we introduce the frequently used notations, terminology, and some fundamental concepts including PageRank, personalized PageRank, heat kernel PageRank, as well as the problem definitions of local graph clustering and network embedding.

Chapter 3 conducts a detailed survey on the local graph clustering, homogeneous network embedding and attributed network embedding. In this chapter, we first survey

existing solutions for local graph clustering. Two HKPR-based local graph clustering algorithms, *i.e.*, **ClusterHKPR** and **HK-Relax** are introduced in detail and their deficiencies and limitations are pointed out. Meanwhile, other relevant methodologies for local graph clustering are introduced. We also present a thorough discussion about the differences between our proposed methods and their similar techniques for computing approximate PPR. Second, we review existing techniques for homogeneous network embedding. We mainly categorize these abundant methods into four types: random walk-based methods, deep learning-based methods, matrix factorization-based methods and other methods. Then, we introduce each method briefly and pinpoint their limitations in terms of either efficiency or effectiveness. Finally, a plethora of existing solutions for attributed network embedding are summarized. We first categorize them into three types, *i.e.*, factorization-based methods, auto-encoder-based methods and other methods, and further briefly overview these methods and point out their deficiencies and limitations.

In Chapter 4, we present **TEA** and **TEA+**. We first introduce Heat Kernel-based local clustering problem and present our solution overview for it. Then, we propose our first solution, **TEA**, which first generates a rough estimation of heat kernel PageRank by performing a deterministic graph traversal and then refine the result with some random walks. After that, we further improve **TEA** to **TEA+** in terms of efficiency by employing two non-trivial optimizations. Finally, we experimentally evaluate **TEA** and **TEA+** against all competing methods in terms of running time and clustering quality on both small and large real-world or synthetic graphs to show the efficiency and effectiveness of our proposed methods. In addition, sensitivity analysis regarding subgraph characteristics and the heat constant parameter is experimentally studied.

In Chapter 5, we demonstrate **NRP**. We first present our novel baseline HNE solution via PPR. Afterward, we propose our node-reweighting scheme and combine it with the baseline to obtain our optimized solution **NRP**. Finally, we compare **NRP** with 13 recent methods and 3 classic methods on 6 benchmark datasets, and also show the practical effectiveness of **NRP** in link prediction, node classification and graph reconstruction tasks as well as its efficiency on large graphs with up to 1.2 billion edges by extensive empirical results. The additional experimental results for parameter analysis, link prediction on evolving graphs and scalability tests are also included.

In Chapter 6, we introduce PANE. First, we propose forward and backward node-attribute PageRank, and further, based on which formulate our optimization objective for attributed network embedding problem. Then, we present PANE, which involves two stages, *i.e.*, approximating forward and backward node-attribute PageRank, and joint factorization of forward and backward node-attribute PageRank matrices. Subsequently, we devise parallel PANE for processing large-scale graphs. Finally, we compare PANE against 10 existing solutions over 8 real datasets in terms of effectiveness in graph mining tasks including link prediction, node classification and attribute inference, as well as efficiency. We also analyze the parameter sensitivities of PANE.

Chapter 7 concludes this thesis and outlines some potential future work in several directions.

Chapter 2

Preliminaries

In this chapter, we first introduce some fundamental concepts regarding graphs and frequently used notations in this thesis. Subsequently, we present the formal definitions of PageRank, personalized PageRank and heat kernel PageRank, respectively. Finally, we present the problem statements of local graph clustering, homogeneous network embedding and attributed network embedding, which will be studied in Chapter 4, Chapter 5, and Chapter 6, respectively.

2.1 Notations and Terminology

Let $G = (V, E)$ be a graph, where V and E denote the node and edge sets, respectively. A graph G is directed if its edges in E are directed, or undirected if its edges in E do not have direction (a.k.a are bidirectional). In a directed graph G , we use $N_{in}(v)$ (resp. $N_{out}(v)$) and $d_{in}(v)$ (resp. $d_{out}(v)$) to denote the set of in-neighbors (resp. out-neighbors) and the in-degree (resp. out-degree) of node v , respectively. Moreover, in an undirected graph G , we use $N(v)$ and $d(v)$ to represent the set of neighbors and the degree of node v , respectively. We denote matrices in bold uppercase, *e.g.*, $\mathbf{M}$. We use $\mathbf{M}[v_i]$ to denote the v_i -th row vector of $\mathbf{M}$, and $\mathbf{M}[:, r_j]$ to denote the r_j -th column vector of $\mathbf{M}$. In addition, we use $\mathbf{M}[v_i, r_j]$ to denote the element at the v_i -th row and r_j -th column of $\mathbf{M}$. Given an index set S , we let $\mathbf{M}[S]$ (resp. $\mathbf{M}[:, S]$) be the matrix block of $\mathbf{M}$ that contains the row (resp. column) vectors of the indices in S . Let $\mathbf{A}$ be the adjacency matrix of the input graph G , *i.e.*, $\mathbf{A}[v_i, v_j] = 1$ if $(v_i, v_j) \in E_V$, otherwise $\mathbf{A}[v_i, v_j] = 0$. Let $\mathbf{D}$ be the diagonal out-degree matrix of G , *i.e.*, $\mathbf{D}[v_i, v_i] = \sum_{v_j \in V} \mathbf{A}[v_i, v_j]$. We define the

Table 2.1: Frequently used notations.

Notation	Description
$G=(V, E)$	A graph with node set V and edge set E .
n, m	The numbers of nodes and edges in G , respectively.
$d_{in}(v)$	The in-degree of node v
$d_{out}(v)$	The out-degree of node v
$N_{out}(v)$	The set of out-neighbors of node v .
$N_{in}(v)$	The set of in-neighbors of node v .
$N(v)$	The set of neighbors of node v
$d(v)$	The degree of node v
$\mathbf{A}, \mathbf{D}, \mathbf{P}$	The adjacency, diagonal degree, and random walk matrices of G .
α	The random walk stopping probability.

random walk matrix (a.k.a. *probability transition matrix*) of G is defined as $\mathbf{P} = \mathbf{D}^{-1}\mathbf{A}$. Accordingly, $\mathbf{P}[s, v] = \frac{1}{|N_{out}(s)|}$ if $(s, v) \in E$, and $\mathbf{P}^k[s, v]$ denotes the probability that a k -hop ($k \geq 1$) random walk from node s would end at v .

In addition to the topological structure of the network, there are often attributes accompanied with nodes in real-world graphs which are referred to as *attributed networks*. The formal definition of *attributed network* is as follows.

Definition 2.1 (Attributed Network). *An attributed network $G = (V, E_V, R, E_R)$ is a graph, consisting of (i) a node set V with cardinality n , (ii) a set of edges E_V of size m , each connecting two nodes in V , (iii) a set of attributes R with cardinality d , and (iv) a set of node-attribute associations E_R , where each element is a tuple $(v_i, r_j, w_{i,j})$ signifying that node $v_i \in V$ is directly associated with attribute $r_j \in R$ with a weight $w_{i,j}$ (i.e., the attribute value).*

2.2 PageRank and Personalized PageRank

PageRank is proposed and implemented in Google [40] to rank web pages, which to a certain degree plays an important role in the ascent of Google to prominence. The PageRank value of node v , i.e., $\pi(v)$, measures the global importance of node v in the graph. The basic idea of PageRank is that a node is important if it has more neighbors. More specifically, if a node v with PageRank value $\pi(v)$ has $N_{out}(v)$ out-neighbors, each

of these neighbors gets $\frac{\pi(v)}{|N_{out}(v)|}$ PageRank value, and thus, the PageRank value of a node v is the sum of the PageRank values received from v 's in-neighbors. Initially, it sets the PageRank value of each node v in the graph to $\frac{1}{n}$, *i.e.*, $\pi(v) = \frac{1}{n}$. After sending and receiving PageRank values over the graph for a large number of iterations, the PageRank values converge as a PageRank vector $\boldsymbol{\pi}$ (*i.e.*, reach a stationary distribution) that satisfies

$$\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}. \quad (2.1)$$

Personalized PageRank [68, 78] (a.k.a. random walk with restart [132]) is the personalized version of PageRank, which reflects the importance of a target node v from the perspective of a source node s in the graph. Consider the following random walk model, in which a walker travels along the graph. Given a start node s and a stopping probability α ($0 \leq \alpha \leq 1$), the walker, at each step, either (i) terminates at the current node with α probability, or (ii) proceeds to a randomly selected out-neighbor of the current node with the remaining $1 - \alpha$ probability. For any node $v \in V$, the PPR $\pi(s, v)$ of v with respect to s is then the probability that the walker from s stops at v . Let $(x_0, x_1, \dots, x_\ell)$ be the nodes that the walker from $x_0 = s$ visits and the length of the random walk be ℓ ($\ell \sim \text{Geometric}(\alpha)$). Here by $\ell \sim \text{Geometric}(\alpha)$ we mean that the length ℓ is picked as L with probability $\mathbb{P}[\ell = L] = \alpha(1 - \alpha)^L$. Thus, the PPR value of node v with respect to source node s is defined as

$$\pi(s, v) = \mathbb{P}[x_\ell = v]. \quad (2.2)$$

We introduce an equivalent definition of PPR. In contrast to PageRank vector (Equation 2.1), in which each entry is initially set to $\frac{1}{n}$, PPR vector with respect to source node s , *i.e.*, $\boldsymbol{\pi}_s$, is initialized as a personalized vector $\mathbf{e}_s$, which is a length- n vector having value 1 at entry s and 0 everywhere else. Similarly, the PPR vector with respect to source node s , *i.e.*, $\boldsymbol{\pi}_s$, satisfies the recursive equation

$$\boldsymbol{\pi}_s = \alpha \mathbf{e}_s + (1 - \alpha) \boldsymbol{\pi}_s \mathbf{P}. \quad (2.3)$$

Using a power series expansion [5] and the convergence of Neumann series, we derive another equivalent form of the PPR vector $\boldsymbol{\pi}_s$:

$$\boldsymbol{\pi}_s = (\mathbf{I} - (\alpha \mathbf{e}_s + (1 - \alpha) \mathbf{P}))^{-1} = \sum_{\ell=0}^{\infty} \alpha(1 - \alpha)^\ell \mathbf{e}_s \mathbf{P}^\ell. \quad (2.4)$$

Furthermore, we can represent the PPR matrix as:

$$\mathbf{\Pi} = \sum_{\ell=0}^{\infty} \alpha(1-\alpha)^\ell \mathbf{P}^\ell, \quad (2.5)$$

where $\mathbf{\Pi}[s] = \boldsymbol{\pi}_s$ and $\mathbf{\Pi}[s, v] = \pi(s, v)$.

2.3 Heat Kernel PageRank

Heat Kernel PageRank is first introduced in [18] as a variant of PPR, and then is shown both theoretically and empirically effective in graph clustering in sequential work [21, 22, 36].

Given a *heat constant* t and two nodes u and v , the HKPR value from u to v is defined as the probability that a random walk of length k starting from u would end at v , where k is sampled from the following Poisson distribution:

$$\eta(k) = \frac{e^{-t} t^k}{k!}. \quad (2.6)$$

Let s be the seed node for local clustering. We define the HKPR vector $\boldsymbol{\rho}_s$ of s as an n -size vector, such that the i -th element of $\boldsymbol{\rho}_s$ equals the HKPR value from s to the i -th node in G . In addition, we use $\boldsymbol{\rho}_s[v]$ to denote the HKPR value from s to v , which is defined by

$$\boldsymbol{\rho}_s[v] = \sum_{k=0}^{\infty} \eta(k) \cdot \mathbf{P}^k[s, v]. \quad (2.7)$$

It can be observed that the only difference between PPR and HKPR is that the length of random walk in PPR is sampled from a geometric distribution $Geometric(\alpha)$, while that in HKPR is sampled from a Poisson distribution with mean t . Although PPR and HKPR have similar definitions, they incur very different challenges in the computation and approximation, as discussed in Section 3.1.4 of Chapter 3.

2.4 Local Graph Clustering

A *cluster* in the undirected graph G is a node-set $S \subseteq V$. Intuitively, a good cluster should be both internally cohesive and well separated from the remainder of G . We say that S is a high-quality cluster if it has a small *conductance* [10] $\Phi(S)$, defined as:

Figure 2.1: An example social graph G for local graph clustering.

$$\Phi(S) = \frac{|\text{cut}(S)|}{\min\{\text{vol}(S), \text{vol}(V \setminus S)\}}, \quad (2.8)$$

where $\text{vol}(S)$ is the *volume* of S , namely, the sum of the degrees of all nodes in S , and $\text{cut}(S)$ is the *cut* of S , *i.e.*, the set of edges with one endpoint in S and another not in S .

Graph clustering [120] is a fundamental problem in graph mining and analysis. However, most classic graph clustering algorithms require touching the entire graph at least once, which can be computationally expensive for massive graphs. Motivated by this, local graph clustering algorithms have received significant interest due to the fact that they find high-quality clusters in a graph with running time proportional to the size of the cluster found and rather than the size of the entire graph. Specifically, the formal definition of local graph clustering is as follows.

Definition 2.2 (Local Graph Clustering [128]). *Given an undirected graph G and a seed node s , a local graph clustering algorithm aims to identify a well-connected cluster S around s , such that $\Phi(S)$ is minimal and the running time is proportional to the size of the cluster it outputs regardless the size of the input graph G .*

Figure 2.1 shows an example of local graph clustering. In Figure 2.1, G is a social graph with 11 nodes, in which each node represents a user and each edge represents a friendship. Given G and a user s as inputs, a local graph clustering algorithm explores the vicinity of user s in G and returns the users in the blue circle, *i.e.*, users v_1, v_2, v_3, v_4 and user s , as a community (*i.e.*, local cluster) S for user s . There are three edges (*i.e.*, friendships) crossing S 's border, *i.e.*, (v_3, v_6) , (v_2, v_8) and (v_2, v_9) , and the sum of degrees (*i.e.*, the total number of friends) of all users in S is 19. By Equation (2.8), the conductance of community S , *i.e.*, $\Phi(S) = \frac{3}{\min\{19, 15\}} = 0.2$, which is minimal among the conductances of all possible communities (*i.e.*, local clusters) containing user s .

Figure 2.2: An example social graph G for HNE.

The first local graph clustering algorithm is proposed by Spielman and Teng in [42], which is based on a random walk model. Subsequently, numerous local graph clustering algorithms including random walk-based methods [3, 21, 128, 190], evolving set process-based methods [4, 54] and flow-based methods [46, 107, 137, 138] are proposed in order to either improve the efficiency as well as the quality (*i.e.*, conductance) of the cluster returned. In Section 3.1 of Chapter 3, we review the related work on local graph clustering.

2.5 Network Embedding

Given a graph G , *network embedding* aims to map each node $v \in V$ to a compact, fixed-length feature vector, thereby facilitating the downstream applications by feeding the embeddings to machine learning or deep learning models. In recent years, a plethora of embedding techniques are proposed for various types of graphs, such as *homogeneous network embedding* [61, 110, 129], *attributed network embedding* [67, 101, 161], *heterogeneous network embedding* [16, 33, 123], *knowledge graph embedding* [80, 146, 153], *uncertain network embedding* [74] and *signed network embedding* [84, 151, 176]. In this thesis, we mainly concentrate on HNE and ANE problems, which receives significant attention from both industry and academia [90, 154, 189]. First, we present the formal definition of the HNE problem.

Definition 2.3 (Homogeneous Network Embedding [110]). *Given a graph $G = (V, E)$, where $|V| = n$ and $|E| = m$, homogeneous network embedding aims to learn an embedding vector $\mathbf{X}[v_i] \in \mathbb{R}^k$ ($k \ll n$) for each node $v_i \in V$ such that it can preserve the topological features surrounding v_i in G .*

To illustrate, we present an example of HNE in the following. Figure 2.2 shows a social graph G with 6 nodes, in which each node represents a user and each edge represents a friendship. Given G and embedding dimensionality $k = 3$ as inputs, an HNE algorithm maps each node to a 3-dimensional embedding vector, *e.g.*, $\mathbf{X}[v_1] = [2.75, 0, 1.8]$, $\mathbf{X}[v_3] = [0, 0.76, 1.05]$ and $\mathbf{X}[v_6] = [2.63, 0.56, 0.66]$ for users v_1, v_3 and v_6 , respectively. From Figure 2.2, it can be observed that user v_1 and user v_6 are friends and have two common friends (*i.e.*, users v_4, v_5), user v_1 and user v_3 are not friends and have only one common friend (*i.e.*, user v_3), while user v_3 and user v_6 are totally strangers. Intuitively, the above observation implies that user v_1 is closer to user v_6 than user v_3 and user v_3 is closer to user v_1 than user v_6 based on the social graph topology. From the embedding vectors, we can observe that the inner products satisfy $\mathbf{X}[v_1] \cdot \mathbf{X}[v_6]^\top = 8.42 > \mathbf{X}[v_1] \cdot \mathbf{X}[v_3]^\top = 1.89 > \mathbf{X}[v_3] \cdot \mathbf{X}[v_6]^\top = 1.12$, which accords with the above intuition and demonstrates that the embedding vectors well preserve the topological features surrounding the users.

In the above example, we see that HNE only concentrates on preserving the topological features in the graph G . If we were to recommend a friend to user v_3 in G , it is more likely to be v_1 than v_6 based on the HNE results. However, in real-world social graphs, users are usually associated with multiple attributes, *e.g.*, gender, age, location and hobbies. For example, if both user v_3 and user v_6 in G are boys of eighteen years old, in the same high-school, super fans of NBA and share many other interests, while user v_1 is from a totally different community, a good social network service would recommend user v_6 as a suggested friend to user v_3 with the consideration of their attributes. However, this can not be done in HNE due to that HNE methods do not preserve the information of node attributes in the embedding vectors. Towards this end, we formally define the ANE problem as follows.

Definition 2.4 (Attributed Network Embedding [161]). *Given an attributed network $G = (V, E_V, R, E_R)$, where $|V| = n$ and $|R| = d$, attributed network embedding aims to learn an embedding vector $\mathbf{X}[v_i] \in \mathbb{R}^k$ ($k \ll \min\{d, n\}$) for each node $v_i \in V$ such that it can preserve both the topological features and attribute information surrounding v_i in G .*

In Section 3.2 and Section 3.3 of Chapter 3, we survey and categorize existing techniques on HNE and ANE, respectively.

Chapter 3

Literature Review

In this chapter, we first review the algorithms for local graph clustering that are most related to our solution, as well as the deficiencies they suffer from in Section 3.1. Afterward, in Section 3.2, we survey the related work on homogeneous network embedding, which is categorized into three main types, *i.e.*, random walk-based methods, deep learning-based methods and matrix factorization-based methods. In addition, an analysis of the limitations of these methods is given. Section 3.3 presents existing work on attributed network embedding, as well as their drawbacks in terms of both effectiveness and efficiency.

3.1 Local Graph Clustering

In this section, we first review two HKPR algorithms, ClusterHKPR and HK-Relax, that are most related to our solutions; after that, we review other work related to local clustering and HKPR computation. Table 3.1 lists the theoretical guarantees of our solutions against ClusterHKPR and HK-Relax in terms of accuracy and time complexity.

3.1.1 ClusterHKPR

ClusterHKPR [21] is a random-walk-based method for computing approximate HKPR. Given a seed node s , it performs $\frac{16 \log n}{\epsilon^3}$ random walks from s , with the constraint that each walk has a length at most K ; after that, for each node v , it uses the fraction of walks that end at v as an estimation $\hat{\rho}_s[v]$ of v 's HKPR. It is shown that with probability

Table 3.1: Theoretical guarantees of our solutions against that of the state-of-the-art solutions.

Algorithm	Accuracy Guarantee	Time Complexity
ClusterHKPR [21]	with probability at least $1 - \epsilon$, $\begin{cases} \hat{\rho}_s[v] - \rho_s[v] \leq \epsilon \cdot \rho_s[v], & \text{if } \rho_s[v] > \epsilon \\ \hat{\rho}_s[v] - \rho_s[v] \leq \epsilon, & \text{otherwise,} \end{cases}$	$O\left(\frac{t \log(n)}{\epsilon^3}\right)$
HK-Relax [36]	$\frac{1}{d(v)} \hat{\rho}_s[v] - \rho_s[v] < \epsilon_a$	$O\left(\frac{te^t \log(\frac{1}{\epsilon_a})}{\epsilon_a}\right)$
Our solutions	with probability at least $1 - p_f$, $\begin{cases} \frac{1}{d(v)} \hat{\rho}_s[v] - \rho_s[v] \leq \epsilon_r \cdot \frac{\rho_s[v]}{d(v)}, & \text{if } \frac{\rho_s[v]}{d(v)} > \delta \\ \frac{1}{d(v)} \hat{\rho}_s[v] - \rho_s[v] \leq \epsilon_r \cdot \delta, & \text{otherwise,} \end{cases}$	$O\left(\frac{t \log(\frac{n}{p_f})}{\epsilon_r^2 \delta}\right)$

at least $1 - \epsilon$, ClusterHKPR guarantees that

$$\begin{cases} |\hat{\rho}_s[v] - \rho_s[v]| \leq (1 + \epsilon) \cdot \rho_s[v], & \text{if } \rho_s[v] > \epsilon \\ |\hat{\rho}_s[v] - \rho_s[v]| \leq \epsilon, & \text{otherwise.} \end{cases}$$

Note that for the above guarantee to be meaningful, $\epsilon \ll 1$ should hold; otherwise, the successful probability $1 - \epsilon$ of ClusterHKPR would be too small, and there could be too many nodes having a large absolute error up to ϵ . In particular, if we are to ensure that ClusterHKPR achieves (d, ϵ_r, δ) -approximation with probability at least $1 - p_f$, then we have to set $\epsilon \leq \min\{\epsilon_r \cdot \delta, p_f\}$. However, when $\epsilon \ll 1$, ClusterHKPR incurs excessive computation cost, since its expected time complexity is inversely proportional to ϵ^3 (see Table 3.1).

3.1.2 HK-Relax

HK-Relax [36] is a deterministic algorithm that runs in $O\left(\frac{te^t \log(\frac{1}{\epsilon_a})}{\epsilon_a}\right)$ time and returns an approximate HKPR vector $\hat{\rho}_s$ such that $\left|\frac{\hat{\rho}_s[v]}{d(v)} - \frac{\rho_s[v]}{d(v)}\right| \leq \epsilon_a$ for any $v \in V$ (see Table 3.1). HK-Relax is similar to our HK-Push algorithm in that they both (i) maintain an approximation HKPR vector $\hat{\rho}_s$ and a number of residue vectors, and (ii) incrementally refine $\hat{\rho}_s$ by applying push operations according to the residue of each node. However, there exist three major differences between HK-Relax and HK-Push. First, HK-Relax and HK-Push define the residue of each node in different manners, due to which HK-Relax requires more sophisticated approaches than HK-Push to update $\hat{\rho}_s$ and the residue vectors after each

push operation. Second, HK-Relax ignores all k -hop residues with $k > 2t \log \frac{1}{\epsilon_a}$, whereas HK-Push retains all residues generated for the refinement of $\hat{\rho}_s$ via random walks. Third, HK-Relax and HK-Push have different termination conditions. Due to these differences, we are unable to combine HK-Relax with random walks to achieve the same performance guarantee provided by our TEA and TEA+ algorithms.

3.1.3 Other Methods for Local Clustering

The first local graph clustering algorithm, *Nibble*, is introduced in the seminal work [42, 128] by Spielman and Teng. After that, Anderson *et al.* [3] propose PR-*Nibble*, a local clustering algorithm based on *personalized PageRank* [68, 78], and show that it improves over *Nibble* in terms of the theoretical guarantees of both clustering quality and time complexity. In turn, Anderson *et al.*'s method is improved in subsequent work [4, 54] based on the *volume-biased evolving set process* [32]. Subsequent work [46, 107, 137, 138, 190] achieves further improved guarantees on the quality of local clustering, but the methods proposed are mostly of theoretical interests only, as they are difficult to implement and offer rather poor practical efficiency. As a consequence, HK-Relax remains the state-of-the-art for local clustering in terms of practical performance [6, 22, 36].

In recent work [41], Shun *et al.* study parallel implementations for *Nibble*, PR-*Nibble*, ClusterHKPR, and HK-Relax, respectively, and are able to achieve significant speedup on a machine with 40 cores. We believe that our algorithms may also exploit parallelism for higher efficiency, but a thorough treatment of this problem is beyond the scope of this thesis.

In addition, many other types of algorithms for finding clusters locally have been proposed, including local modularity maximization [25], minimum degree maximization [29, 127], local spectral methods [69, 70, 95, 100], and motif-based methods [135, 171, 186]. These methods either use other clustering metrics or do not provide any quality guarantees for the output clusters.

3.1.4 Methods for Personalized PageRank

We note that TEA and TEA+ are similar in spirit to several recent methods [39, 45, 47, 48] for computing personalized PageRank, since they all combine a push-operation-based

algorithm with random walks. Hence, at first glance, it may seem that we can simply adopt and extend these techniques to address HKPR computation. Unfortunately, this is not the case as HKPR is inherently more sophisticated than PPR. In particular, even though both HKPR and PPR measure the proximity of a node v with respect to another node u by the probability that a random walk starting from u would end at v , they differ significantly in the ways that they define random walks:

- PPR’s random walks are *Markovian*: in each step of a walk, it terminates with a fixed probability $\alpha \in (0, 1)$, regardless of the previous steps.
- HKPR’s random walks are *non-Markovian*: the termination probability of a walk at the i -th step is a function of i , i.e., the walk has to remember the number of steps that it has traversed, so as to decide whether it should terminate.

The Markovianess of PPR random walks is a key property exploited in the methods in [39, 45, 47, 48]. Specifically, when computing the PPR $p(u, v)$ from node u to node v , the methods in [39, 45, 48] require performing a *backward search* which starts from v and traverses the incoming edges of each node in a backward manner. For each node w encountered and each of its incoming neighbor x , the backward search needs to calculate the probability that a random walk hitting x at a certain step would arrive at w at the next step. For PPR random walks, this probability is a constant decided only by α and the number of x ’s outgoing neighbors. Unfortunately, for HKPR random walks the probability is not a constant, but a variable depending on the number of steps that the walk has taken before reaching x . In other words, this variable is not unique even when w and x are fixed, due to which the backward search no longer can be utilized. This issue makes it unpalatable to extend the methods in [39, 45, 48] to compute HKPR.

Meanwhile, the FORA method in [47] does not require a backward search; instead, it combines random walks with a forward search from u that is similar to the **HK-Push** algorithm used in **TEA**. However, **TEA** is more sophisticated than FORA as it needs to account for the non-Markovianess of HKPR, and there are three major differences between the two methods. First, **TEA** requires maintaining multiple residue vectors in its invocation of **HK-Push**, since it needs to differentiate the residues generated at different steps of the forward search; otherwise, it would not be able to combine the results

of HK-Push with random walks because of HKPR’s non-Markovianness. In contrast, FORA only needs to maintain one residue vector, as the Markovianness of PPR allows it to merge the residues produced at different steps of the forward search. Second, the theoretical analysis of TEA is more challenging than that of FORA, since it is more complicated to (i) maintain and update multiple residue vectors and (ii) combine random walks with the forward traversal in a way that takes into account the non-Markovianness of HKPR. Third, TEA provides an accuracy guarantee in terms of each node’s *degree-normalized* HKPR, whereas FORA’s accuracy guarantee is on each node’s PPR without normalization.

Last but not the least, we note that our TEA+ algorithm, which significantly improves over TEA in terms of practical efficiency, is based on a new optimization that is specifically designed for HKPR and our notion of (d, ϵ_r, δ) -approximation. This optimization is inapplicable for PPR computation, which further differentiates TEA+ from FORA.

3.2 Homogeneous Network Embedding

Network embedding is a hot topic in graph mining, for which there exists a large body of literature as surveyed in [12, 28, 178]. In what follows, we review HNE methods that are most relevant to this work.

3.2.1 Random Walk-based Methods

A classic methodology for HNE computation is to learn embeddings from random walk simulations, which is inspired by the success of Word2Vec embedding in natural language processing [103]. Earlier methods in this category include DeepWalk [110], LINE [130], node2vec [61] and Walklets [112]. The basic idea is to learn the embedding of a node v by iteratively “pulling” the embeddings of positive context nodes (*i.e.*, those that are on the random walks originating from v) towards that of v , and “pushing” the embeddings of negative context nodes (*i.e.*, the nodes that are not connected to v) away from v . Subsequent proposals [17, 117] construct a multilayer graph over the original graph G , and then perform random walks on different layers to derive more effective embeddings. Recent techniques APP [185] and VERSE [134] improve the quality of their embeddings

by refining the procedures for generating and learning from random walks samples. Instead of using a predefined sampling distribution, **SeedNE** [52] adaptively sample negative context nodes in terms of their informativeness.

The main problem of random-walk-based HNE learning in general is their immense computational costs (proportional to the number of random walks), which can be prohibitive for large graphs. The high running time could be reduced with massively-parallel hardware, *e.g.*, in **PBG** [91], and/or with GPU systems, *e.g.*, in **Graphy** [191]. Nevertheless, they still incur a high financial cost for consuming large amounts of computational resources.

3.2.2 Deep Learning-based Methods

HNEs can also be learned directly from the graph structure using a deep neural network, without performing random walks. Training such a deep neural network, however, also incurs very high computational overhead, especially for large graphs [134]. Notably, **SDNE** [142] and **DNGR** [14] employ multi-layer auto-encoders with a target proximity matrix to learn embeddings. **GAE** [86] combines the graph convolutional network [85] and auto-encoder models to learn embeddings. **PRUNE** [88] utilizes a Siamese neural network to preserve both pointwise mutual information and global PageRank of nodes. **NetRA** [175] and **DRNE** [136] learn embeddings by feeding node sequences to a long short-term memory (LSTM) model. **DVNE** [188] learns a Gaussian distribution in the Wasserstein space with the deep variational model as the latent representation of each node. **GA** [1] applies graph attention mechanism to a closed-form expectation of the limited random-walk co-occurrence matrices [110] to learn the embeddings. **GraphGAN** [143] and **ANE** [30] adopt the popular generative adversarial networks (GAN) to accurately model the vertex connectivity probability. As demonstrated in our experiments, none of these methods scale to large graphs.

3.2.3 Matrix Factorization-based Methods

Another popular methodology for HNE computation is through factorizing a *proximity matrix* $\mathbf{M} \in \mathbb{R}^{n \times n}$, where n is the number nodes in the input graph G , and each entry $\mathbf{M}[i, j]$ signifies the *proximity* between a pair of nodes $v_i, v_j \in G$. The main research

question here is how to choose an appropriate $\mathbf{M}$ that (i) captures the graph topology well and (ii) is efficient to compute and factorize on large graphs. Specifically, to satisfy (i), each entry $\mathbf{M}[i, j] \in \mathbf{M}$ should accurately reflect the proximity between nodes v_i, v_j via indirect connections, which can be long and complex paths. Meanwhile, to satisfy (ii) above, the computation / factorization of $\mathbf{M}$ should be done in memory. This means that $\mathbf{M}$ should either be sparse, or can be efficiently factorized without materialization. In addition, note that for a directed graph G , the notion of proximity is also directed, meaning that it is possible that $\mathbf{M}[i, j] \neq \mathbf{M}[j, i]$. Thus, methods that rely on the property that $\mathbf{M}$ is symmetric are limited to undirected graphs.

Earlier factorization-based work, including [2, 13, 131, 152], directly computes $\mathbf{M}$ before factorizing it to obtain node embeddings. For instance, spectral embedding [131] simply outputs the top k eigenvectors of the normalized Laplacian matrix of an undirected graph G as node embeddings. This method has limited effectiveness [61, 110], as the Laplacian matrix only captures one-hop connectivity information for each node. To remedy this problem, one idea is to construct a higher-order proximity matrix $\mathbf{M}$ to capture multi-hop connectivity for each node [13, 152, 162]. However, such a higher-order proximity matrix $\mathbf{M}$ is usually no longer sparse; consequently, materializing $\mathbf{M}$ becomes prohibitively expensive for large graphs due to the $O(n^2)$ space complexity for n nodes.

Recent work [108, 182, 183] construct network embeddings without materializing $\mathbf{M}$. Many of these methods, however, rely on the assumption that $\mathbf{M}$ is symmetric; consequently, they are limited to undirected graphs as discussed above. For instance, AROPE [183] first applies an eigen-decomposition on the adjacency matrix $\mathbf{A}$ of an undirected graph G , and then utilizes the decomposition results to derive each node's embedding to preserve proximity information, without explicitly constructing $\mathbf{M}$. Similar approaches have been adopted in [108, 182]. In particular, RandNE [182] uses a Gaussian random projection of $\mathbf{M}$ directly as node embeddings without factorization to achieve high efficiency, at the cost of lower result effectiveness.

The authors of [115] prove that random-walk-based methods such as DeepWalk, LINE and node2vec essentially perform matrix factorizations. Thus, they propose NetMF, which factorize a proximity matrix $\mathbf{M}$ that approximates the closed form representation of DeepWalk's implicit proximity matrix. However, NetMF requires materializing a dense

$\mathbf{M}$, which is infeasible for large graphs. **NetSMF** [114] improves the efficiency of **NetMF** by sparsifying $\mathbf{M}$ using the theory of spectral sparsification. However, **NetSMF** is still rather costly as it requires simulating a large number of random walks to construct $\mathbf{M}$. **GRA** [98] iteratively fine-tunes the proximity matrix to obtain enhanced result effectiveness, at the expense of high computational costs. Lastly, a concurrent work **STRAP** [172] explicitly constructs and factorizes a PPR-based proximity matrix, which we elaborate towards the end of this section.

3.2.4 Other Methods

There also exist several techniques that generate embeddings without random walks, neural networks or matrix factorization. In particular, **NetHiex** [99] applies expectation maximization to learn embeddings that capture the neighborhood structure of each node, as well as the latent hierarchical taxonomy in the graph. **RaRE** [62] considers both the proximity and popularity of nodes, and derive embeddings by maximizing a posteriori estimation using stochastic gradient descent. **GraphWave** [34] represents each node’s network neighborhood via a low-dimensional embedding by leveraging heat wavelet diffusion patterns, so as to capture structural roles of nodes in networks. **node2hash** [145] transplants the feature hashing technique for word embeddings to embed nodes in networks. A common problem with the above methods is that they do not aim to preserve proximity information between nodes; consequently, they are generally less effective for tasks such as link prediction and graph reconstruction, as we demonstrate in our experiments in Section 6.4.

3.3 Attributed Network Embedding

ANE is a popular topic and there exists a large body of literature. Here we mainly category them into three types based the methods they adopt.

3.3.1 Factorization-based methods

Given an attributed network G with n nodes, existing factorization-based methods mainly involve two stages: (i) build a proximity matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$ that models the proximity

between nodes based on graph topology or attribute information; (ii) factorize $\mathbf{M}$ via techniques such as SGD [11], ALS [26], and coordinate descent [156]. Specifically, TADW [161] constructs a second-order proximity matrix $\mathbf{M}$ based on the adjacency matrix of G , and aims to reconstruct $\mathbf{M}$ by the product of the learned embedding matrix and the attribute matrix. HSCA [177] ensures that the learned embedding vectors of connected nodes are close in the embedding space. AANE [75] constructs a proximity matrix $\mathbf{M}$ using the cosine similarities between the attribute vectors of nodes. BANE [164] learns a binary embedding vector per node, *i.e.*, $\{-1, 1\}^k$, by minimizing the reconstruction loss of a unified matrix that incorporates both graph topology and attribute information. BANE reduces space overheads at the cost of accuracy. To further balance the trade-off between space cost and representation accuracy, LQANR [163] learns embedding vectors $\in \{-2^b, \dots, -1, 0, 1, \dots, 2^b\}^k$, where b is the bit-width. All these factorization-based methods incur immense overheads in building and factorizing the $n \times n$ proximity matrix. Further, these methods are designed for undirected graphs only.

3.3.2 Auto-encoder-based methods

An auto-encoder [58] is a neural network model consisting of an encoder that compresses the input data to obtain embeddings and a decoder that reconstructs the input data from the embeddings, with the goal to minimize the reconstruction loss. Existing methods either use different proximity matrices as inputs or design various neural network structures for the auto-encoder. Specifically, ANRL [181] combines auto-encoder with SkipGram model to learn embedding vectors. DANE [51] designs two auto-encoders to reconstruct the high-order proximity matrix and the attribute matrix respectively. ARGAN [109] integrates auto-encoder with graph convolutional networks [85] and generative adversarial networks [59] together. STNE [97] samples nodes via random walks and feeds the attribute vectors of the sampled nodes into a LSTM-based auto-encoder [71]. NetVAE [81] compresses the graph structures and node attributes with a shared encoder for transfer learning and information integration. CAN [101] embeds both nodes and attributes into two Gaussian distributions using a graph convolutional network and a dense encoder. None of these auto-encoder-based methods considers edge directions. Further, they suffer from severe efficiency issues due to the expensive training process of auto-encoders.

SAGE2VEC [121] proposed an enhanced auto-encoder model that preserves global graph structure and meanwhile handles the non-linearity and sparsity of both graph structure and attributes. AdONE [8] proposed an auto-encoder model for detecting and minimizing the effect of community outliers while generating embeddings.

3.3.3 Other methods

PRRE [187] categorizes node relationships into positive, ambiguous and negative types, according to the graph and attribute proximities between nodes, and then employs Expectation Maximization (EM) [31] to learn embedding vectors. SAGE [67] samples and aggregates features from a node's local neighborhood and learns embedding vectors by LSTM and pooling. NetHash [159] builds a rooted tree for each node by expanding along the neighborhood of the node, and then recursively sketches the rooted tree to get a summarized attribute list as the embedding vector of the node. PGE [73] groups nodes into clusters based on their attributes, and then trains neural networks with biased neighborhood samples in clusters to generate embedding vectors. ProGAN [53] adopts generative adversarial networks to generate node proximities, followed by neural networks to learn node embedding vectors from the generated node proximities. DGI [139] derives embedding vectors via graph convolutional networks, such that the mutual information between the embedding vectors for nodes and the embedding vector for the whole graph is maximized. MARINE [157] preserves the long-range spatial dependencies between nodes into embeddings by minimizing the information discrepancy in a Reproducing Kernel Hilbert Space.

Recently, there are embedding studies on attributed heterogeneous networks that consist of not only graph topology and node attributes, but also node types and edge types. When there are only one type of node and one type of edge, these methods effectively work on attributed networks. For instance, Alibaba proposed GATNE [15], to process attributed heterogeneous network embedding. For each node on every edge type, it learns an embedding vector, by using SkipGram model and random walks over the attributed heterogeneous network. Then it obtains the overall embedding vector for each node by concatenating the embedding vectors of the node over all edge types. GATNE incurs expensive training overheads and highly relies on the power of distributed systems.

Chapter 4

Local Graph Clustering via Heat Kernel PageRank¹

In this chapter, we study the local graph clustering on undirected graphs. Given an undirected graph G and a *seed* node s , the *local clustering* problem aims to identify a high-quality cluster containing s in time roughly proportional to the size of the cluster, regardless of the size of G . This problem finds numerous applications on large-scale graphs. Recently, *heat kernel PageRank*, which is a variant of *personalized PageRank* and a measure of the proximity of nodes in graphs, is applied to this problem and found to be more efficient compared with prior methods. However, existing solutions for computing HKPR either are prohibitively expensive or provide unsatisfactory error approximation on HKPR values, rendering them impractical especially on billion-edge graphs. Thus, we present TEA and TEA+, two novel local graph clustering algorithms based on HKPR, to address the aforementioned limitations. Specifically, these algorithms provide non-trivial theoretical guarantees in *relative error* of HKPR values and the time complexity. The basic idea is to utilize deterministic graph traversal to produce a rough estimation of exact HKPR vector, and then exploit Monte-Carlo random walks to refine the results in an optimized and non-trivial way. In particular, TEA+ offers practical efficiency and effectiveness due to non-trivial optimizations. Extensive experiments on real-world datasets demonstrate that TEA+ outperforms the state-of-the-art algorithm by more than four times on most benchmark datasets in terms of computational time when achieving

¹The work in this chapter has been published as [170] in SIGMOD 2019.

Table 4.1: Frequently used notations.

Notation	Description
$\bar{d}$	The average degree of the graph, <i>i.e.</i> , $\frac{2m}{n}$
t	The heat constant of HKPR
$\eta(k), \psi(k)$	See Equation (2.6) and Equation (4.1), respectively
$\rho_s[v]$	HKPR of v w.r.t. s , defined by Equation (2.7)
ϵ_r, δ, p_f	Parameters of an approximate HKPR, as in Section 4.1.2
$\mathbf{r}_s^{(k)}[v]$	The k -hop residue of v during performing push operations from s
$\mathbf{q}_s[v]$	The reserve of v during performing push operations from s
K	The maximum number of hops during performing push operations from the seed node

the same clustering quality, and in particular, is an order of magnitude faster on large graphs including the widely studied *Twitter* and *Friendster* datasets.

The remainder of the chapter is organized as follows. In Section 4.1, we present the problem definition, as well as an overview of our solution objective. Section 4.2 and Section 4.3 present TEA and TEA+, respectively. We then conduct a comprehensive evaluation of the practical efficiency of our algorithms against the competitors in Section 4.4. Section 4.5 briefly summarizes this chapter. Table 4.1 lists some frequently used notations in this chapter.

4.1 Problem Formulation

In this section, we first formally define the *heat kernel PageRank-based local cluster* problem, followed by an overview of the high-level idea and objective of our solution.

4.1.1 Heat-Kernel-PageRank-based Local Clustering

According to Section 2.4 of Chapter 2, given an undirected graph G and a seed node s , *local graph clustering* only explores a local neighborhood of s and aims to find a cluster S surrounding s such that the cluster S has the minimal conductance, which is defined as, the ratio of the number of its edge cuts to the number of its total edges (see Equation 2.8 in Section 2.4 of Chapter 2). Let s be the seed node for local clustering, and the HKPR vector ρ_s of s as an n -size vector, such that the i -th element of ρ_s equals the

HKPR value from s to the i -th node in G (see its definition in Section 2.3 of Chapter 2). Existing heat-kernel-based algorithms [21, 23, 36, 41] all adopt a two-phase approach. In particular, they first compute an *approximate* HKPR vector $\hat{\rho}_s$ for s , and then perform a *sweep* as follows:

1. Take the set S^* of nodes with non-zero values in $\hat{\rho}_s$.
2. Sort the nodes $v \in S^*$ in descending order of $\frac{\hat{\rho}_s[v]}{d(v)}$. Let S_i^* be a set containing the first i nodes in the sorted sequence.
3. Inspect S_i^* in ascending order of i . Return the set S_i^* with the smallest conductance among the ones that have been inspected.

It is shown in [41, 190] that the above sweep can be conducted in $O(|S^*| \cdot \log |S^*|)$ time, assuming that $\hat{\rho}_s$ is given in a sparse representation with $O(|S^*|)$ entries. In contrast, the computation of $\hat{\rho}_s$ is much more costly, and hence, has been the main subject of research in existing work [21, 23, 36, 41].

4.1.2 Solution Objective

Our solution for local clustering is based on heat kernel PageRank, and it follows the same two-phase framework in the existing work [18, 19, 21, 23, 36, 41]. That is, we also compute an approximate HKPR vector $\hat{\rho}_s$ for s , and then conduct a sweep on $\hat{\rho}_s$. However, we require that $\hat{\rho}_s$ should be a (d, ϵ_r, δ) -approximate HKPR vector, which is a criterion not considered in any existing work [18, 19, 21, 23, 36, 41].

Definition 4.1 ((d, ϵ_r, δ) -approximate HKPR). *Let ρ_s be the HKPR vector for a seed node s , and $\hat{\rho}_s$ be an approximated version of ρ_s . $\hat{\rho}_s$ is (d, ϵ_r, δ) -approximate if it satisfies the following conditions:*

- For every $v \in V$ with $\frac{\rho_s[v]}{d(v)} > \delta$,

$$\left| \frac{\hat{\rho}_s[v]}{d(v)} - \frac{\rho_s[v]}{d(v)} \right| \leq \epsilon_r \cdot \frac{\rho_s[v]}{d(v)};$$

- For every $v \in V$ with $\frac{\rho_s[v]}{d(v)} \leq \delta$,

$$\left| \frac{\hat{\rho}_s[v]}{d(v)} - \frac{\rho_s[v]}{d(v)} \right| \leq \epsilon_r \cdot \delta.$$

□

In other words, we require $\frac{\hat{\rho}_s[v]}{d(v)}$ to provide a relative error guarantee when $\frac{\rho_s[v]}{d(v)} > \delta$, and an absolute error guarantee when $\frac{\rho_s[v]}{d(v)} \leq \delta$. This is to ensure that when we sort the nodes in descending order or $\frac{\hat{\rho}_s[v]}{d(v)}$ (which is a crucial step in the sweep for local clustering), the sorted sequence would be close to the one generated based on $\frac{\rho_s[v]}{d(v)}$. We do not consider relative error guarantees when $\frac{\rho_s[v]}{d(v)} \leq \delta$, because (i) ensuring a small relative error for such a node v requires an extremely accurate estimation of its normalized HKPR, which would incur significant computation overheads, and (ii) providing such high accuracy for v is unnecessary, since v 's tiny normalized HKPR value indicates that it is not relevant to the result of local clustering.

By the definition of HKPR (in Equation (2.7)), the HKPR value of v w.r.t. s is a weighted sum of k -hop random walk transition probabilities from s to v , where k is a Poisson distributed length. Thus, a straightforward method to compute (d, ϵ_r, δ) -approximate HKPR for seed node s is to conduct Monte-Carlo simulations using a large number of random walks. Specifically, each random walk should start from s , and should have a length that is sampled from the Poisson distribution in Equation (2.6). Let n_r be the total number of random walks, and $\hat{\rho}_s[v]$ be the fraction of walks that end at a node v . Then, we can use $\hat{\rho}_s[v]$ as an estimation of $\rho_s[v]$. By the Chernoff bound (see Lemma 4.5) and union bound, it can be shown that $\hat{\rho}_s$ is (d, ϵ_r, δ) -approximate with probability at least

$$1 - n \cdot \exp\left(-\frac{n_r \cdot \epsilon_r^2 \cdot \delta}{2(1 + \epsilon_r/3)}\right).$$

Therefore, if we are to ensure that the above probability is at least $1 - p_f$, then we can set $n_r = \frac{2(1+\epsilon_r/3) \log(\frac{n}{p_f})}{\epsilon_r^2 \cdot \delta}$. In that case, the time required to generate the random walks is $O\left(\frac{t \log(\frac{n}{p_f})}{\epsilon_r^2 \cdot \delta}\right)$. The main issue of this Monte-Carlo approach, however, is that it incurs considerable overheads in practice (see our experimental results in Section 4.4.4). To explain, consider a node v with a small $\rho_s[v]$ but a relatively large $\frac{\rho_s[v]}{d(v)} > \delta$. By the requirements of (d, ϵ_r, δ) -approximation, we need to ensure that $|\hat{\rho}_s[v] - \rho_s[v]| \leq \epsilon_r \cdot \rho_s[v]$. In turn, this requires that the number n_r of random walks should be large; otherwise, the number of walks that end at v would be rather small, in which case the estimation of $\rho_s[v]$ would be inaccurate.

We also note that none of the existing methods [21, 36] can be adopted to compute (d, ϵ_r, δ) -approximate HKPR efficiently. In particular, HK-Relax [36] only ensures that

for any node $v \in V$, $\frac{1}{d(v)} |\hat{\rho}_s[v] - \rho_s[v]| < \epsilon_a$. If we are to use HK-Relax for (d, ϵ_r, δ) -approximation, then we need to set $\epsilon_a = \epsilon_r \cdot \delta$, in which case its complexity would be $O\left(\frac{te^t \log\left(\frac{1}{\epsilon_r \cdot \delta}\right)}{\epsilon_r \cdot \delta}\right)$, which is considerably worse than the time complexity of the Monte-Carlo approach, due to the exponential term e^t . The ClusterHKPR [21] algorithm suffers from a similar issue, as we point out in Chapter 3.

To mitigate the deficiencies of the aforementioned methods, we present (in Section 4.2 and Section 4.3) two more efficient HKPR algorithms that satisfy the following requirements:

1. Return a (d, ϵ_r, δ) -approximate HKPR vector $\hat{\rho}_s$ with at least $1 - p_f$ probability, where p_f is a user-specified parameter;
2. Run in $O\left(\frac{t \log\left(\frac{n}{p_f}\right)}{\epsilon_r^2 \cdot \delta}\right)$ expected time.

4.2 The TEA Algorithm

This section presents TEA², our first-cut solution for (d, ϵ_r, δ) -approximate HKPR. TEA is motivated by the inefficiency of the Monte-Carlo approach explained in Section 4.1.2, *i.e.*, it requires a large number of random walks to accurately estimate HKPR values. To address this issue, we propose to combine Monte-Carlo with a secondary algorithm that could help reduce the number of random walks needed. In particular, we first utilize the secondary algorithm to efficiently compute a rough estimation $\mathbf{q}_s[v]$ of $\rho_s[v]$, and then perform random walks to refine each $\mathbf{q}_s[v]$, so as to transform $\mathbf{q}_s$ into a (d, ϵ_r, δ) -approximate HKPR vector $\hat{\rho}_s$. Towards this end, there are several issues that we need to address:

1. How to design a secondary algorithm that could generate a rough approximation of the HKPR vector at a small computation cost?
2. How to enable Monte-Carlo to leverage the output of the secondary algorithm for improved efficiency?

²Two-Phase Heat Kernel Approximation

Algorithm 1: HK-Push

Input: Graph G , seed node s , threshold r_{max}
Output: An approximate HKPR vector $\mathbf{q}_s$ and $K + 1$ residue vectors $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$

```

1  $\mathbf{q}_s \leftarrow \mathbf{0}, \mathbf{r}_s^{(k)} \leftarrow \mathbf{0}$  for  $k = 0, \dots$ ;
2  $\mathbf{r}_s^{(0)}[s] \leftarrow 1$ ;
3 while  $\exists v \in V, k$  such that  $\mathbf{r}_s^{(k)}[v] > r_{max} \cdot d(v)$  do
4      $\mathbf{q}_s[v] \leftarrow \mathbf{q}_s[v] + \frac{\eta(k)}{\psi(k)} \cdot \mathbf{r}_s^{(k)}[v]$ ;
5     for  $u \in N(v)$  do
6          $\mathbf{r}_s^{(k+1)}[u] \leftarrow \mathbf{r}_s^{(k+1)}[u] + \left(1 - \frac{\eta(k)}{\psi(k)}\right) \cdot \frac{\mathbf{r}_s^{(k)}[v]}{d(v)}$ ;
7      $\mathbf{r}_s^{(k)}[v] \leftarrow 0$ ;
8  $K \leftarrow \max \left\{ k \mid \exists v \in V, \mathbf{r}_s^{(k)}[v] > 0 \right\}$ ;
9 return  $\mathbf{q}_s$  and  $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ ;
    
```

3. How to ensure that the combination of Monte-Carlo and the secondary algorithm still provides strong theoretical assurance in terms of time complexity and accuracy?

To answer the above questions, we first present our choice of the secondary algorithm, referred to as HK-Push, in Section 4.2.1; after that, we elaborate the integration of HK-Push and Monte-Carlo in Section 4.2.2, and then provide a theoretical analysis of the combined algorithm in Section 4.2.3.

4.2.1 HK-Push

Algorithm 1 shows the pseudo-code of our secondary algorithm, HK-Push, for deriving a rough approximation $\hat{\rho}_s$ of the HKPR vector. Its basic idea is to begin with a vector $\hat{\rho}_s$ where $\hat{\rho}_s[s] = 1$ and $\hat{\rho}_s[v] = 0$ for all nodes v except s , and then perform a traversal of G starting from s , and keep refining $\hat{\rho}_s$ during the course of the traversal. In addition, to facilitate its combination with random walks, it not only returns an approximate HKPR vector $\mathbf{q}_s$, but also outputs $K + 1$ auxiliary vectors $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)} \in \mathbb{R}^n$ that could be used to guide the random walks conducted by Monte-Carlo. We refer to $\mathbf{q}_s$ as the *reserve vector* and $\mathbf{r}_s^{(k)}$ as the *k-hop residue vector*. Accordingly, for any node v , $\mathbf{q}_s[v]$ and $\mathbf{r}_s^{(k)}[v]$ are referred to as the *reserve* and *k-hop residue* of v , respectively.

More specifically, HK-Push takes as input G , s , and a residue threshold r_{max} . It begins by setting all entries in $\mathbf{q}_s$ and $\mathbf{r}_s^{(k)}$ to zero, except that $\mathbf{r}_s^{(0)}[s] = 1$ (Lines 1-2). After

that, it starts an iterative process to traverse G from s (Lines 3-7). In particular, in each iteration, it inspects the k -hop residue vectors to identify a node v whose k -hop residue $\mathbf{r}_s^{(k)}[v]$ is above $r_{max} \cdot d(v)$. If such a node v exists, then the algorithm updates the reserve and k -hop residue of v , as well as the $(k+1)$ -hop residues of v 's neighbors. In particular, it first adds $\frac{\eta(k)}{\psi(k)}$ fraction of v 's k -hop residue $\mathbf{r}_s^{(k)}[v]$ to its reserve $\mathbf{q}_s[v]$, where $\eta(k)$ is as defined in Equation (2.6) and

$$\psi(k) = \sum_{\ell=k}^{\infty} \eta(\ell), \quad (4.1)$$

and then evenly distribute the other $1 - \frac{\eta(k)}{\psi(k)}$ fraction to the $(k+1)$ -hop residues of v 's neighbors (Lines 4-6). For convenience, we refer to the operation of distributing a fraction of v 's k -hop residue to one of its neighbors as a *push* operation. The rationale of the aforementioned push operations is that, if a random walk from s arrives at v at the k -th hop, then it has $\frac{\eta(k)}{\psi(k)}$ probability to terminate at v , and has $1 - \frac{\eta(k)}{\psi(k)}$ probability to traverse to a randomly selected neighbor of v at the next hop. After that, the algorithm sets $\mathbf{r}_s^{(k)}[v] = 0$ (Line 7), and proceeds to the next iteration. After the iterative process terminates, HK-Push identifies the largest K such that $\mathbf{r}_s^{(K)}$ has at least one non-zero entry, and returns $\mathbf{q}_s$ and $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$. The following lemma shows a crucial property of these reserve and residue vectors:

Lemma 4.1. *Consider any iteration in Algorithm 1. Let $\mathbf{q}_s$ and $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ be the reserve and residue vectors constructed by the end of the iteration. We have*

$$\rho_s[v] = \mathbf{q}_s[v] + \sum_{u \in V} \sum_{k=0}^K \mathbf{r}_s^{(k)}[u] \cdot \mathbf{h}_u^{(k)}[v], \quad (4.2)$$

where

$$\mathbf{h}_u^{(k)}[v] = \sum_{\ell=0}^{\infty} \frac{\eta(k+\ell)}{\psi(k)} \cdot \mathbf{P}^\ell[u, v], \quad (4.3)$$

i.e., $\mathbf{h}_u^{(k)}[v]$ is the probability that a random walk stops at v , conditioned on the k -hop of the walk is at u . $\square$

Proof. This proof is based on induction. Given reserve vector $\mathbf{q}_s$ and residue vectors $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ constructed by the end of any iteration in Algorithm 1, we define $\mathbf{f}_s[v]$ as

$$\mathbf{f}_s[v] = \mathbf{q}_s[v] + \sum_{u \in V} \sum_{k=0}^K \mathbf{r}_s^{(k)}[u] \cdot \mathbf{h}_u^{(k)}[v].$$

First, let us consider the initial condition, in which all entries of $\mathbf{r}_s^{(0)}$ are zero, except $\mathbf{r}_s^{(0)}[s] = 1$, and other vectors are zero vectors. This implies that

$$\begin{aligned} \mathbf{f}_s[v] &= \mathbf{q}_s[v] + \sum_{u \in V} \sum_{k=0}^K \left[\mathbf{r}_s^{(k)}[u] \cdot \sum_{\ell=0}^{\infty} \frac{\eta(k+\ell)}{\psi(k)} \mathbf{P}^\ell[u, v] \right] \\ &= \sum_{\ell=0}^{\infty} \eta(\ell) \cdot \mathbf{P}^\ell[s, v] = \boldsymbol{\rho}_s[v]. \end{aligned}$$

Namely, Equation (4.2) holds in the initial case.

Furthermore, we assume that after j iterations, reserve vector $\mathbf{q}_s$ and residue vectors $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ satisfy Equation (4.2), namely $\mathbf{f}_s[v] = \boldsymbol{\rho}_s[v]$. In the beginning of the $(j+1)$ -th iteration, we will conduct push operations with an entry (w, i) , where $w \in V$ and $i < K$. Then, we define the change of $\mathbf{f}_s[v]$ after this iteration as

$$\begin{aligned} \Delta(w, i) &= \left(\widehat{\mathbf{q}}_s[v] + \sum_{u \in V} \sum_{k=0}^K \widehat{\mathbf{r}}_s^{(k)}[u] \cdot \mathbf{h}_u^{(k)}[v] \right) \\ &\quad - \left(\mathbf{q}_s[v] + \sum_{u \in V} \sum_{k=0}^K \mathbf{r}_s^{(k)}[u] \cdot \mathbf{h}_u^{(k)}[v] \right), \end{aligned}$$

where $\widehat{\mathbf{q}}_s$ is the updated reserve vector, $\widehat{\mathbf{r}}_s^{(i)}$ and $\widehat{\mathbf{r}}_s^{(i+1)}$ are the updated residue vectors after performing push operations in this iteration. Then, by Lines 4-7 of Algorithm 1, we have

$$\widehat{\mathbf{q}}_s[w] - \mathbf{q}_s[w] = \mathbf{r}_s^{(i)}[w] \cdot \frac{\eta(i)}{\psi(i)},$$

$$\widehat{\mathbf{r}}_s^{(i)}[w] - \mathbf{r}_s^{(i)}[w] = -\mathbf{r}_s^{(i)}[w],$$

and $\forall u \in N(w)$, $\widehat{\mathbf{r}}_s^{(i+1)}[u] - \mathbf{r}_s^{(i+1)}[u] = \left(1 - \frac{\eta(i)}{\psi(i)}\right) \cdot \mathbf{r}_s^{(i)}[w] \cdot \mathbf{P}[w, u]$. Hence, the change of $\mathbf{f}_s[v]$ after this iteration is

$$\begin{aligned} \Delta(w, i) &= (\widehat{\mathbf{q}}_s[v] - \mathbf{q}_s[v]) + (\widehat{\mathbf{r}}_s^{(i)}[w] - \mathbf{r}_s^{(i)}[w]) \cdot \mathbf{h}_w^{(i)}[v] \\ &\quad + \sum_{u \in N(w)} (\widehat{\mathbf{r}}_s^{(i+1)}[u] - \mathbf{r}_s^{(i+1)}[u]) \cdot \mathbf{h}_u^{i+1}[v]. \end{aligned}$$

Algorithm 2: k -RandomWalk

Input: Graph G , node u , constant k ,
Output: An end node v

```

1  $\ell \leftarrow 0$ ;
2  $v_0 \leftarrow u$ ;
3 while True do
4   if  $\text{rand}(0, 1) \leq \frac{\eta(k+\ell)}{\psi(k+\ell)}$  then
5     break;
6   else
7     Pick a neighbor  $v_{\ell+1} \in N(v_\ell)$  uniformly at random;
8      $\ell \leftarrow \ell + 1$ ;
9 return  $v_\ell$ ;
    
```

Notice that

$$\begin{aligned}
 & \sum_{u \in \mathcal{N}(w)} (\hat{\mathbf{r}}_s^{(i+1)}[u] - \mathbf{r}_s^{(i+1)}[u]) \cdot \mathbf{h}_u^{i+1}[v] \\
 &= \left(1 - \frac{\eta(i)}{\psi(i)}\right) \mathbf{r}_s^{(i)}[w] \cdot \sum_{u \in \mathcal{N}(w)} \mathbf{P}[w, u] \cdot \mathbf{h}_u^{i+1}[v] \\
 &= \mathbf{r}_s^{(i)}[w] \sum_{u \in \mathcal{N}(w)} \left[\mathbf{P}[w, u] \sum_{l=0}^{\infty} \frac{\eta(i+1+l)}{\psi(i)} \mathbf{P}^l[u, v] \right] \\
 &= \mathbf{r}_s^{(i)}[w] \cdot \left(\mathbf{h}_w^{(i)}[v] - \frac{\eta(i)}{\psi(i)} \right).
 \end{aligned}$$

Therefore, $\Delta(w, i) = \hat{\mathbf{q}}_s[v] - \mathbf{q}_s[v] - \mathbf{r}_s^{(i)}[w] \cdot \mathbf{h}_w^{(i)}[v] + \mathbf{r}_s^{(i)}[w] \cdot \left(\mathbf{h}_w^{(i)}[v] - \frac{\eta(i)}{\psi(i)} \right) = 0$, which implies $\mathbf{f}_s[v]$ is invariant and still equals $\boldsymbol{\rho}_s[v]$ after the $(j+1)$ -th iteration. This completes the proof of this lemma. $\square$

Intuitively, Lemma 4.1 indicates that for any node v , $\mathbf{q}_s[v]$ is a lower bound of $\boldsymbol{\rho}_s[v]$ in any iteration in Algorithm 1. Since each iteration of Algorithm 1 only increases the reserve of a selected node and never decreases any others, it guarantees that the difference between $\mathbf{q}_s$ and $\boldsymbol{\rho}_s$ monotonically decreases, i.e., $\mathbf{q}_s$ becomes a better approximation of $\boldsymbol{\rho}_s$ as the algorithm progresses. Although HK-Push may produce results that are far from satisfying the requirements of (d, ϵ_r, δ) -approximate HKPR, it is sufficient for the integration with Monte-Carlo, as we show in Section 4.2.2.

Algorithm 3: TEA

Input: Graph G , seed node s , thresholds ϵ_r and δ , threshold r_{max} , and failure probability p_f

Output: A (d, ϵ_r, δ) -approximate HKPR vector $\hat{\rho}_s$

```

1 if  $\sum_{v \in V} p_f^{d(v)-1} \leq 1$  then
2    $p'_f \leftarrow p_f$ ;
3 else
4    $p'_f \leftarrow \frac{p_f}{\sum_{v \in V} p_f^{d(v)-1}}$ ;
5  $\omega \leftarrow \frac{2(1+\epsilon_r/3) \log(1/p'_f)}{\epsilon_r^2 \delta}$ ;
6  $(\hat{\rho}_s, \mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}) \leftarrow \text{HK-Push}(s, r_{max})$ ;
7  $\alpha \leftarrow \sum_{k=0}^K \sum_{u \in V} \mathbf{r}_s^{(k)}[u]$ ;
8  $n_r \leftarrow \alpha \cdot \omega$ ;
9 for  $i = 1$  to  $n_r$  do
10   Sample an entry  $(u, k)$  from  $V \times \{0, 1, \dots, K\}$  with probability  $\frac{\mathbf{r}_s^{(k)}[u]}{\alpha}$ ;
11    $v \leftarrow k\text{-RandomWalk}(G, u, k)$ ;
12    $\hat{\rho}_s[v] \leftarrow \hat{\rho}_s[v] + \frac{\alpha}{n_r}$ ;
13 return  $\hat{\rho}_s$ ;
```

4.2.2 Algorithm

Basic Idea. To incorporate HK-Push into Monte-Carlo, we utilize Equation (4.2), which shows that the exact HKPR vector ρ_s can be expressed as a function of $\mathbf{q}_s$, $\mathbf{r}_s^{(k)}$, and $\mathbf{h}_u^{(k)}[v]$ for any $u, v \in V$, and $k \in [0, K]$. Recall that $\mathbf{q}_s$ and $\mathbf{r}_s^{(k)}$ are outputs of HK-Push, while $\mathbf{h}_u^{(k)}[v]$ is the conditional probability that a random walk terminates at node v given that its k -th hop is at node u . If we can accurately estimate $\mathbf{h}_u^{(k)}[v]$ for any given u, v , and k , then we can combine the estimated values with $\mathbf{q}_s$ and $\mathbf{r}_s^{(k)}$ to obtain an approximate version of ρ_s . Towards this end, we conduct random walks starting from u , and estimate $\mathbf{h}_u^{(k)}[v]$ based on the frequency that v appears at the k -th hop of the random walks. Algorithm 2 shows the pseudo-code of our random walk generation method, referred to as k -RandomWalk.

The following lemma proves that k -RandomWalk samples each node v with probability $\mathbf{h}_u^{(k)}[v]$.

Lemma 4.2. *Given G , u , and k , for any node v , Algorithm 2 returns v with probability $\mathbf{h}_u^{(k)}[v]$.*

Proof. Recall that Algorithm 2 consists of several iterations (see Lines 3-8). In the ℓ -th ($\ell = 0, 1, \dots$) iteration, the algorithm terminates with $\frac{\eta(k+\ell)}{\psi(k+\ell)}$ probability (Line 4); with the other $1 - \frac{\eta(k+\ell)}{\psi(k+\ell)}$ probability, it samples a neighbor node of current node and set the sampled node as current node (Lines 6-8). Let v_i be the node at the beginning of the i -th iteration, and $p(v, \ell)$ be the probability that the algorithm terminates at the ℓ -th iteration with v as the returned node. We will prove that, given $G, s = u, k$ as the input to Algorithm 2, we have

$$p(v, \ell) = \frac{\eta(k+\ell)}{\psi(k)} \cdot \mathbf{P}^\ell[u, v]. \quad (4.4)$$

Note that if Equation (4.4) holds, then the overall probability that v is sampled as returned node is

$$\sum_{\ell=0}^{\infty} p(v, \ell) = \sum_{\ell=0}^{\infty} \frac{\eta(k+\ell)}{\psi(k)} \cdot \mathbf{P}^\ell[u, v] = \mathbf{h}_u^{(k)}[v],$$

which establishes the lemma.

We prove by induction. For the base case where $\ell = 0$, we have $v = s$, and the probability that the 0-th iteration of Algorithm 2 terminates is $\frac{\eta(k)}{\psi(k)}$. In that case, $p(v, \ell) = \frac{\eta(k)}{\psi(k)}$ if $v = u$; otherwise $p(v, \ell) = 0$. Meanwhile, it can be verified that when $\ell = 0$, the r.h.s of Equation (4.4) equals $\frac{\eta(k)}{\psi(k)}$ if $v = u$, and 0 otherwise. Therefore, Equation (4.4) holds when $\ell = 0$.

Assume that Equation (4.4) holds when $\ell = i$. Then, for any node $w \in V$, the probability that w is at the beginning of i -th iteration is

$$p(w, \ell) \cdot \frac{\psi(k+i)}{\eta(k+i)} = \frac{\psi(k+i)}{\psi(k)} \cdot \mathbf{P}^i[u, w].$$

Now consider the $(i+1)$ -th iteration. Since Algorithm 2 does not terminate at i -th iteration, for any node v , the probability that v is picked as current node is

$$\begin{aligned} & \left(1 - \frac{\eta(k+i)}{\psi(k+i)}\right) \cdot \frac{\psi(k+i)}{\psi(k)} \cdot \sum_{w \in V} \mathbf{P}^i[u, w] \cdot \mathbf{P}[w, v] \\ &= \frac{\psi(k+i+1)}{\psi(k)} \cdot \mathbf{P}^{i+1}[u, v]. \end{aligned}$$

In the $(i+1)$ -th iteration, Algorithm 2 terminates with probability $\frac{\eta(k+i+1)}{\psi(k+i+1)}$, which implies

that v is returned with probability

$$\begin{aligned} p(v, i+1) &= \frac{\eta(k+i+1)}{\psi(k+i+1)} \cdot \frac{\psi(k+i+1)}{\psi(k)} \cdot \mathbf{P}^{i+1}[u, v] \\ &= \frac{\eta(k+i+1)}{\psi(k)} \cdot \mathbf{P}^{i+1}[u, v]. \end{aligned}$$

Therefore, the lemma is proved. $\square$

Details. Algorithm 3 illustrates the pseudo-code of TEA, our first-cut solution that (i) incorporates HK-Push and k -RandomWalk and (ii) computes a (d, ϵ_r, δ) -approximate HKPR vector $\hat{\rho}_s$ with at least $1 - p_f$ probability for any given seed node s . Given G , ϵ_r , δ , r_{max} , and failure probability p_f as inputs, the algorithm starts by invoking HK-Push with three parameters: G , s , and r_{max} (Line 6). It returns an approximate HKPR vector $\hat{\rho}_s$ and $K+1$ residue vectors $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ from HK-Push. Then, TEA proceeds to refine $\hat{\rho}_s$ using k -RandomWalk (Lines 7-13). In particular, TEA first computes the sum α of the residues in $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ (Line 7), and computes

$$\omega = \frac{2(1 + \epsilon_r/3) \log(1/p'_f)}{\epsilon_r^2 \cdot \delta},$$

where

$$p'_f = \begin{cases} p_f, & \text{if } \sum_{v \in V} p_f^{d(v)-1} \leq 1 \\ \frac{p_f}{\sum_{v \in V} p_f^{d(v)-1}}, & \text{otherwise.} \end{cases} \quad (4.5)$$

Note that p'_f can be pre-computed when the graph G is loaded. Given ω , TEA performs $n_r = \alpha \cdot \omega$ random walks using k -RandomWalk (Lines 9-12), such that the starting point u of each walk is sampled with probability $\frac{\mathbf{r}_s^{(k)}[u]}{\alpha}$ (Line 10). Note that this sampling procedure can be conducted efficiently by conducting an *alias structure* [141] on the non-zero elements in $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$. For each random walk that ends at a node v , TEA increases $\hat{\rho}_s[v]$ by $\frac{\alpha}{n_r}$ (Line 12).

Observe that the parameter r_{max} in TEA controls the balance between its two main components: HK-Push and k -RandomWalk. In particular, by Algorithm 1, the entries in $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ are bounded by r_{max} . Therefore, when r_{max} is small, $\alpha = \sum_{k=0}^K \sum_{u \in V} \mathbf{r}_s^{(k)}[u]$ would decrease, in which case the total number $\alpha \cdot \omega$ of random walks conducted by TEA would also be small. As a trade-off, the processing cost of HK-Push would increase, as shown in the following lemma:

Lemma 4.3. *Given residue threshold r_{max} , Algorithm 1 runs in $O\left(\frac{1}{r_{max}}\right)$ time and requires $O\left(\frac{1}{r_{max}}\right)$ space (excluding the space required by the input graph). In addition, in the residue vectors $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ returned by Algorithm 1, there are $O\left(\frac{1}{r_{max}}\right)$ non-zero elements in total.*

Proof. Since the initialization of vectors in Algorithm 1 can be implemented in $O(1)$ cost by using *hashmap*, the total cost of Algorithm 1 is mainly determined by the number of push operations performed before termination. Recall that in each iteration of Algorithm 1 (Lines 4-7), if there exists an entry (u, k) such that u 's current k -hop residue is greater than $r_{max} \cdot d(u)$, HK-Push will convert $\frac{\eta(k)}{\psi(k)}$ fraction of the current k -hop residue into its reserve. This implies that in each iteration occurred on entry (u, k) , its reserve will be increased by at least $\frac{\eta(k)}{\psi(k)} \cdot r_{max} \cdot d(u)$. Let $\mathbf{r}_s^{(k)}[u]$ denote the total k -hop residue that u receives and $x_{u,k}$ be the number of iterations occurred on entry (u, k) . Then, we have

$$\frac{\eta(k)}{\psi(k)} \cdot r_{max} \cdot d(u) \cdot x_{u,k} \leq \frac{\eta(k)}{\psi(k)} \cdot \mathbf{r}_s^{(k)}[u],$$

and thus $d(u) \cdot x_{u,k} \leq \frac{\mathbf{r}_s^{(k)}[u]}{r_{max}}$. Since in each iteration occurred on entry (u, k) , HK-Push will perform $d(u)$ push operations, the total number of push operations occurred on all (u, k) entries is then bounded by

$$\begin{aligned} n_p &= \sum_{u \in V} \sum_{k=0}^{\infty} d(u) \cdot x_{u,k} \leq \sum_{u \in V} \sum_{k=0}^{\infty} \frac{\mathbf{r}_s^{(k)}[u]}{r_{max}} \\ &= \frac{1}{r_{max}} \cdot \sum_{k=0}^{\infty} \sum_{u \in V} \mathbf{r}_s^{(k)}[u] = \frac{1}{r_{max}}. \end{aligned}$$

Since each push operation on entry (u, k) will increase the value of $\mathbf{r}_s^{(k)}[u]$, the number of non-zero elements in residue vectors $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ returned by Algorithm 1 is bounded by the total number of push operations, i.e., $O\left(\frac{1}{r_{max}}\right)$.

Excluding the space required by the input graph, the space cost of Algorithm 1 is incurred by the reserve vector and the non-zero elements in residue vectors, which is $O\left(\frac{1}{r_{max}}\right)$. Therefore, the lemma is proved. $\square$

To strike a balance between the costs incurred by HK-Push and k -RandomWalk, we set $r_{max} = O\left(\frac{1}{\omega \cdot t}\right)$. In that case, the processing cost of HK-Push is $O(\omega \cdot t)$, while k -RandomWalk incurs $O(\alpha \omega t)$ expected cost, due to the following lemma:

Lemma 4.4. *The expected cost of each invocation of k -RandomWalk is $O(t)$.*

Proof. With regard to each invocation of k -RandomWalk, the time complexity relies on the total number of nodes it visits in the random walk. According to Lemma 4.2, the probability that any node $v \in V$ is returned as the end node at ℓ -th iteration is $p(v, \ell)$. If an invocation of k -RandomWalk terminates at v in ℓ -th iteration, then ℓ nodes are visited before it returns v . Considering all nodes in the graph, we have the expected time of each invocation of k -RandomWalk as follows

$$\begin{aligned}
 \sum_{\ell=0}^{\infty} \sum_{v \in V} p(v, \ell) \cdot \ell &= \sum_{\ell=0}^{\infty} \frac{\eta(k + \ell)}{\psi(k)} \cdot \ell \cdot \sum_{v \in V} \mathbf{P}^{\ell}[u, v] \\
 &= \sum_{\ell=0}^{\infty} \frac{\eta(k + \ell)}{\psi(k)} \cdot \ell \\
 &= \sum_{\ell=0}^{\infty} \left[\frac{\eta(k + \ell)}{\psi(k)} \cdot (k + \ell) - \frac{\eta(k + \ell)}{\psi(k)} \cdot k \right] \\
 &= \frac{1}{\psi(k)} \cdot \left(\sum_{\ell=0}^{\infty} \eta(k + \ell) \cdot (k + \ell) \right) - k \\
 &= \frac{t}{\psi(k)} \cdot \sum_{\ell=k-1}^{\infty} \eta(\ell) - k \\
 &= \frac{t}{\psi(k)} \cdot \psi(k - 1) - k \\
 &= t + t \cdot \frac{\eta(k - 1)}{\psi(k)} - k.
 \end{aligned} \tag{4.6}$$

Note that $\frac{t}{k} \cdot \eta(k - 1) = \eta(k) \leq \sum_{\ell=k}^{\infty} \eta(\ell) = \psi(k)$. Then $t \cdot \frac{\eta(k-1)}{\psi(k)} \leq k$. This implies that Equation (4.6) is no greater than t , which finishes our proof. $\square$

Hence, setting $r_{max} = O(\frac{1}{\omega \cdot t})$ ensures that the overheads of HK-Push and k -RandomWalk are roughly comparable.

4.2.3 Analysis

Correctness. Let $\mathbf{q}_s$ denote the approximate HKPR vector obtained from HK-Push in Line 6 of TEA, and $\hat{\boldsymbol{\rho}}_s$ be the approximate HKPR vector eventually output by TEA. In the following, we show that $\hat{\boldsymbol{\rho}}_s$ is a (d, ϵ_r, δ) -approximate HKPR vector.

First, by Lemma 4.1, we have the following equation for any node v :

$$\boldsymbol{\rho}_s[v] = \mathbf{q}_s[v] + \mathbf{a}_s[v], \quad (4.7)$$

where

$$\mathbf{a}_s[v] = \alpha \cdot \sum_{k=0}^K \sum_{u \in V} \frac{\mathbf{r}_s^{(k)}[u]}{\alpha} \cdot \mathbf{h}_u^{(k)}[v]. \quad (4.8)$$

Consider the i -th invocation of k -RandomWalk in TEA. Let (u, k) be the entry sampled by TEA (in Line 10) before the invocation, and v be the node returned by k -RandomWalk. Let Y_i be a Bernoulli variable that equals 1 if v is returned, and 0 otherwise. By Lemma 4.2,

$$\mathbb{E}[Y_i] = \sum_{u \in V} \sum_{k=0}^K \frac{\mathbf{r}_s^{(k)}[u]}{\alpha} \cdot \mathbf{h}_u^{(k)}[v]. \quad (4.9)$$

Combining Equations (4.8) and (4.9), we have

$$\mathbb{E} \left[\sum_{i=1}^{n_r} Y_i \cdot \frac{\alpha}{n_r} \right] = \mathbf{a}_s[v], \quad (4.10)$$

which indicates that $\sum_{i=1}^{n_r} Y_i \cdot \frac{\alpha}{n_r}$ is an unbiased estimator of $\mathbf{a}_s[v]$. By the Chernoff bound (in Lemma 4.5), we can prove that this estimator is highly accurate, based on which we obtain Theorem 4.1.

Lemma 4.5 (Chernoff Bound [20]). *Let $X_1, X_2, \dots, X_{n_r} \in [0, 1]$ be i.i.d. random variables, and $X = \sum_{i=1}^{n_r} X_i$. Then,*

$$\mathbb{P}[|X - \mathbb{E}[X]| \geq \lambda] \leq \exp \left(-\frac{\lambda^2}{2\mathbb{E}[X] + 2\lambda/3} \right). \quad \square$$

Theorem 4.1. *TEA outputs a (d, ϵ_r, δ) -approximate HKPR vector $\hat{\boldsymbol{\rho}}_s$ with probability at least $1 - p_f$.*

Proof. Let Y_i be as defined in the context of Equation (4.9), and let $Y = \sum_{i=1}^{n_r} Y_i$. By Line 12 of TEA, $\hat{\boldsymbol{\rho}}_s[v] = \mathbf{q}_s[v] + \frac{Y \cdot \alpha}{n_r}$. By Equation (4.10), the expectation of Y is

$$\mathbb{E}[Y] = \mathbb{E} \left[\sum_{i=1}^{n_r} Y_i \right] = \frac{n_r}{\alpha} (\boldsymbol{\rho}_s[v] - \mathbf{q}_s[v]) \leq \frac{n_r}{\alpha} \cdot \boldsymbol{\rho}_s[v].$$

Let $\lambda = \frac{n_r \epsilon_r}{\alpha} \cdot \rho_s[v]$ and n_r be as defined in Line 8 of TEA. By the Chernoff bound (see Lemma 4.5), for any node v in V with $\rho_s[v] > d(v) \cdot \delta$, we have

$$\begin{aligned} \mathbb{P}[|Y - \mathbb{E}[Y]| \geq \lambda] &= \mathbb{P}[|\hat{\rho}_s[v] - \rho_s[v]| \geq \epsilon_r \cdot \rho_s[v]] \\ &\leq \exp\left(-\frac{n_r \cdot \epsilon_r^2 \cdot \rho_s[v]}{2\alpha \cdot (1 + \epsilon_r/3)}\right) \leq (p'_f)^{d(v)}. \end{aligned}$$

On the other hand, let $\lambda = \frac{n_r \epsilon_r \delta d(v)}{\alpha}$. Then, for any node v in V with $\rho_s[v] \leq d(v) \cdot \delta$,

$$\begin{aligned} \mathbb{P}[|Y - \mathbb{E}[Y]| \geq \lambda] &= \mathbb{P}[|\hat{\rho}_s[v] - \rho_s[v]| \geq d(v) \cdot \epsilon_r \delta] \\ &\leq \exp\left(-\frac{n_r \cdot \epsilon_r^2 \delta^2 d^2(v)}{2\alpha(1 + \epsilon_r/3) \cdot \rho_s[v]}\right) \leq (p'_f)^{d(v)}. \end{aligned}$$

By the union bound, for $V_1 = \{v | v \in V \text{ s.t. } \rho_s[v] > d(v) \cdot \delta\}$, we have

$$\mathbb{P}\left[\bigcup_{v \in V_1} \{|\hat{\rho}_s[v] - \rho_s[v]| \geq \epsilon_r \cdot \rho_s[v]\}\right] \leq \sum_{v \in V_1} (p'_f)^{d(v)},$$

and for $V_2 = \{v | v \in V \text{ s.t. } \rho_s[v] \leq d(v) \cdot \delta\}$, we have

$$\mathbb{P}\left[\bigcup_{v \in V_2} \left\{\left|\frac{\hat{\rho}_s[v]}{d(v)} - \frac{\rho_s[v]}{d(v)}\right| \geq \epsilon_r \delta\right\}\right] \leq \sum_{v \in V_2} (p'_f)^{d(v)}.$$

Therefore, we have the following results, respectively. With probability at least $1 - \sum_{v \in V} (p'_f)^{d(v)}$, for every node v in V with $\frac{\rho_s[v]}{d(v)} > \delta$, $\left|\frac{\hat{\rho}_s[v]}{d(v)} - \frac{\rho_s[v]}{d(v)}\right| \leq \epsilon_r \cdot \frac{\rho_s[v]}{d(v)}$, and for every node v in V with $\frac{\rho_s[v]}{d(v)} \leq \delta$, $\left|\frac{\hat{\rho}_s[v]}{d(v)} - \frac{\rho_s[v]}{d(v)}\right| \leq \epsilon_r \delta$.

By the definition of p'_f in Equation (4.5), if $\sum_{v \in V} p_f^{d(v)-1} \leq 1$, then $p'_f = p_f$, which leads to $\sum_{v \in V} (p'_f)^{d(v)} = \sum_{v \in V} p_f^{d(v)} \leq p_f$; otherwise, $p'_f = \frac{p_f}{\sum_{v \in V} p_f^{d(v)-1}}$, hence $\sum_{v \in V} (p'_f)^{d(v)} < \sum_{v \in V} \frac{p_f^{d(v)}}{\sum_{v \in V} p_f^{d(v)-1}} = p_f$. Namely, $\hat{\rho}_s$ is a (d, ϵ_r, δ) -approximate HKPR vector with probability at least $1 - p_f$. $\square$

Time and Space Complexities. Given $r_{max} = O\left(\frac{1}{\omega \cdot t}\right)$, HK-Push runs in $O(\omega \cdot t)$ time and $O(\omega \cdot t)$ space. In addition, the computation of α as well as the construction of alias structure on $\mathbf{r}_s^{(k)}$ can be done in time and space linear to the total number of non-zero entries in $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$, which is $O(\omega \cdot t)$ (see Lemma 4.3). Furthermore, in terms of both space and time, the total expected cost incurred by the random walks in TEA is $O(\alpha \omega t)$, where $\alpha < 1$. Therefore, the time complexity of TEA is

Algorithm 4: HK-Push+

Input: Graph G , seed node s , thresholds ϵ_r and δ , maximum hop number K , maximum number of pushes n_p

Output: An approximate HKPR vector $\mathbf{q}_s$ and $K + 1$ residue vectors $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$

```

1  $\mathbf{q}_s \leftarrow \mathbf{0}, \mathbf{r}_s^{(k)} \leftarrow \mathbf{0}$  for  $k = 0, \dots, K$ ;
2  $\mathbf{r}_s^{(0)}[s] \leftarrow 1$ ;
3  $i \leftarrow 0$ ;
4 while  $\exists v \in V, k < K$  such that  $\mathbf{r}_s^{(k)}[v] > \frac{\epsilon_r \cdot \delta}{K} \cdot d(v)$  do
5    $i \leftarrow i + d(v)$ ;
6   if  $i \geq n_p$  or  $\sum_{\ell=0}^K \max_{u \in V} \left\{ \frac{\mathbf{r}_s^{(\ell)}[u]}{d(u)} \right\} \leq \epsilon_r \cdot \delta$  then
7     break;
    Lines 8-11 are the same as Lines 4-7 in Algorithm 1;
Line 12 is the same as Line 9 in Algorithm 1;
```

$$O\left(\frac{1}{r_{\max}} + \alpha \cdot \omega t\right) = O\left(\frac{t \log(1/p_f')}{\epsilon_r^2 \cdot \delta}\right) = O\left(\frac{t \log(n/p_f)}{\epsilon_r^2 \cdot \delta}\right),$$

and its space complexity is $O\left(n + m + \frac{t \log(n/p_f)}{\epsilon_r^2 \cdot \delta}\right)$, where the $n + m$ term is due to storing of the input graph.

4.3 The TEA+ Algorithm

Although TEA provides a strong accuracy guarantee, we observe in our experiments that it often performs a large number of random walks, which degrades its computation efficiency. One may attempt to reduce the cost of random walks by decreasing the residue threshold $r_{\max}$ in TEA (see the discussion in the end of Section 4.2.2), but this cost reduction would be offset by the fact that HK-Push incurs a larger overhead when $r_{\max}$ is small.

In this section, we present TEA+, an algorithm that significantly improves over TEA in terms of practical efficiency without degrading its theoretical guarantees. TEA+ is similar in spirit to TEA in that it combines random walks with a variant of HK-Push, but there is a crucial difference: after TEA+ obtains the residue vectors $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$, it may reduce some entries in the residue vectors before performing random walks. That is, for each node u that has a non-zero entry $\mathbf{r}_s^{(k)}[u]$ for some k , TEA+ may choose to perform

a smaller number of random walks from u than TEA does, which effectively reduces the total cost of random walks. Establishing the correctness of this pruning approach, however, is non-trivial. In what follows, we first discuss in Section 4.3.1 the extreme case where we can derive (d, ϵ_r, δ) -approximate HKPR while *ignoring all elements in the residue vectors* $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$; after that, in Section 4.3.2, we generalize our discussions to the case where we reduce the non-zero entries in the residue vectors instead of completely omitting them.

4.3.1 The Case without Random Walks

Suppose that we are to let TEA achieve (d, ϵ_r, δ) -approximation without performing any random walks. In that case, we would need to ensure that Line 6 of TEA obtains a (d, ϵ_r, δ) -approximate HKPR vector from HK-Push. Towards this end, we present the following theorem:

Theorem 4.2. *Let $\mathbf{q}_s$ and $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ be the reserve and residue vectors returned by HK-Push. If*

$$\sum_{\ell=0}^K \max_{v \in V} \left\{ \frac{\mathbf{r}_s^{(\ell)}[v]}{d(v)} \right\} \leq \epsilon_a, \quad (4.11)$$

then, for any node v in G , we have

$$\left| \frac{\mathbf{q}_s[v]}{d(v)} - \frac{\rho_s[v]}{d(v)} \right| \leq \epsilon_a. \quad (4.12)$$

Proof. By Lemma 4.6 and the definition of $\mathbf{h}_u^{(k)}[v]$ in Equation 4.3, we have $\frac{\mathbf{h}_u^{(k)}[v]}{d(v)} = \frac{\mathbf{h}_v^{(k)}[u]}{d(u)}$. Then we can rewrite Equation (4.2) as follows

$$\begin{aligned} \rho_s[v] - \mathbf{q}_s[v] &= d(v) \cdot \sum_{u \in V} \sum_{k=0}^K \left[\frac{\mathbf{r}_s^{(k)}[u]}{d(u)} \cdot \mathbf{h}_v^{(k)}[u] \right] \\ &\leq d(v) \cdot \sum_{k=0}^K \left[\max_{u \in V} \left\{ \frac{\mathbf{r}_s^{(k)}[u]}{d(u)} \right\} \cdot \sum_{u \in V} \mathbf{h}_v^{(k)}[u] \right]. \end{aligned}$$

Now we prove that $\sum_{u \in V} \mathbf{h}_v^{(k)}[u] = 1$ for any node $v \in V$ and $k \in [0, K]$.

$$\begin{aligned} \sum_{u \in V} \mathbf{h}_v^{(k)}[u] &= \sum_{\ell=0}^{\infty} \left[\frac{\eta(k+\ell)}{\psi(k)} \sum_{u \in V} \mathbf{P}^\ell[v, u] \right] \\ &= \sum_{\ell=0}^{\infty} \frac{\eta(k+\ell)}{\psi(k)} = 1. \end{aligned}$$

Hence, $\rho_s[v] - \mathbf{q}_s[v] \leq d(v) \cdot \sum_{k=0}^K \max_{u \in V} \left\{ \frac{\mathbf{r}_s^{(k)}[u]}{d(u)} \right\}$. Once Inequality (4.11) held, we have $\frac{\rho_s[v]}{d(v)} - \frac{\mathbf{q}_s[v]}{d(v)} \leq \epsilon_a$, which completes the proof. $\square$

Theorem 4.2 provides a sufficient condition (*i.e.*, Inequality (4.11)) for HK-Push to ensure ϵ_a absolute error in each $\frac{\mathbf{q}_s[v]}{d(v)}$. By Definition 4.1, such $\mathbf{q}_s$ is a (d, ϵ_r, δ) -approximate HKPR vector as long as $\epsilon_a \leq \epsilon_r \cdot \delta$. That said, it is rather inefficient to let HK-Push run until Inequality (4.11) is satisfied. Instead, we propose to let HK-Push run with a fixed budget of processing cost. If it is able to satisfy Inequality (4.11) with $\epsilon_a = \epsilon_r \cdot \delta$ before the budget is depleted, then we return $\mathbf{q}_s$ as the final result; otherwise, we proceed to refine $\mathbf{q}_s$ using random walks (see Chapter 4.3.2).

Based on the above discussion, we present HK-Push+ (in Algorithm 4), which is a revised version of HK-Push with three major changes. First, HK-Push+'s input parameters include three thresholds ϵ_r , δ , and n_p , and it has two new termination conditions (in Line 6): (i) Inequality (4.11) holds with $\epsilon_a = \epsilon_r \cdot \delta$; (ii) The number of push operations that it performs reaches n_p . Recall that a push operation refers to the operation of converting part of a node's k -hop residue to one of its neighbor's $(k+1)$ -hop residue. In other words, HK-Push+ runs in $O(n_p)$ time and requires $O(n_p)$ space, and it returns a (d, ϵ_r, δ) -approximate HKPR vector whenever Inequality (4.11) is satisfied.

Second, HK-Push+ judiciously performs push operations only on nodes v with residue $\mathbf{r}_s^{(k)}[v] > \frac{\epsilon_r \cdot \delta}{K} \cdot d(v)$ (Line 4), whereas HK-Push conducts push operations only when $\mathbf{r}_s^{(k)}[v]$ is larger than an input given threshold $r_{max} \cdot d(v)$. The rationale is that HK-Push+ strives to reduce the k -hop residue of each node below $\frac{\epsilon_r \cdot \delta}{K} \cdot d(v)$, so as to satisfy Inequality (4.11); in contrast, HK-Push is not guided by Inequality (4.11), and hence, uses an ad hoc threshold r_{max} instead.

Third, HK-Push+ makes the maximum number K of hops be specified as an input parameter, whereas HK-Push does not fix K in advance. We use a fixed K in HK-Push+ because (i) as mentioned, Line 4 of HK-Push+ requires testing whether there exists a node v with $\mathbf{r}_s^{(k)}[v] > \frac{\epsilon_r \cdot \delta}{K} \cdot d(v)$, and (ii) such a test can be efficiently implemented when K is fixed. Otherwise, whenever K changes, we would need to recheck all nodes' residues to see if $\mathbf{r}_s^{(k)}[v] > \frac{\epsilon_r \cdot \delta}{K} \cdot d(v)$ holds, which would incur considerable overheads. Meanwhile, HK-Push can afford to let K dynamically change, since it uses a fixed residue threshold r_{max} given as input. In our implementation of HK-Push+, we set

$$K = c \cdot \frac{\log(\frac{1}{\epsilon_r \cdot \delta})}{\log(\bar{d})},$$

where $\bar{d}$ is the average degree of the nodes in G , and c is a constant that we decide based on our experiments in Section 4.4.2.

Recall that K limits the push operations within K hops from the seed node, which is an important parameter controlling the termination of **HK-Push+**. If K is too small, it will lead to a small overhead for **HK-Push+** but a significant cost for k -**RandomWalk**, and *vice versa*. Consequently, it is crucial to find an appropriate value for K so as to strike a good balance between **HK-Push+** and **Monte-Carlo**. In particular, there are two factors we need to take into account when choosing K :

1. K ought to be changed along with $\epsilon_r \cdot \delta$;
2. We need large K for graphs with small degrees, and conversely, small K for large-degree graphs.

This reason is as follows. If K is a fixed constant and small, the number of push operations performed by **HK-Push+** will be bounded by a maximum number. Recall that in each iteration of **HK-Push+**, we will perform $d(v)$ push operations, where $d(v)$ is the number of neighbors of current node v . This implies that the total number of push operations performed within K hops from s on the graph with small average degree will be rather limited compared with that of the graph with high average degree. As a result, if the number of push operations is not large enough, **TEA+** will degrade to **Monte-Carlo** and the overhead for random walks will be significant especially when the value of $\epsilon_r \cdot \delta$ is small. Considering these two factors, we propose to choose K by the following equation:

$$(1/\bar{d})^{\frac{K}{c}} = \epsilon_r \cdot \delta, \tag{4.13}$$

where $\bar{d}$ is the average degree of the input graph. The intuition behind Equation (4.13) is that, after K -hops of push operations, the residues tend to approach $\epsilon_r \cdot \delta$, and hence, the value for α would not be large.

Table 4.2: An example for TEA+

	$k \leq 2$	$k = 3$	$k = 4$
$\sum_{v \in V} \mathbf{r}_s^{(k)}[v]$	0	0.1	0.3
$\max_{v \in V} \frac{\mathbf{r}_s^{(k)}[v]}{d(v)}$	0	$\frac{\mathbf{r}_s^{(k)}[v_1]}{d(v_1)} = 0.0025$	$\frac{\mathbf{r}_s^{(k)}[v_2]}{d(v_2)} = 0.0076$
$\max_{v \in V} \mathbf{r}_s^{(k)}[v]$	0	$\mathbf{r}_s^{(k)}[v_1] = 0.0025$	$\mathbf{r}_s^{(k)}[v_2] = 0.076$

4.3.2 The Case with Random Walks

Suppose that HK-Push+ depletes its computation budget n_p before it satisfies Inequality (4.11) with $\epsilon_a = \epsilon_r \cdot \delta$. In that case, the HKPR vector $\mathbf{q}_s$ returned by HK-Push+ does not ensure (d, ϵ_r, δ) -approximation, and we need to refine $\mathbf{q}_s$ by conducting random walks according to the residue vectors $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ returned by HK-Push+. To reduce the number of random walks required, we propose to reduce the residues in $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ when conducting random walks, based on the following intuition.

First, the reduction of residues would incur some errors in the approximate HKPR vector, as demonstrated in Theorem 4.2 (where we ignore all residues in $\mathbf{r}_s^{(k)}$ and using $\mathbf{q}_s$ directly as the final approximate HKPR vector). Second, if we only reduce the residues in $\mathbf{r}_s^{(k)}$ by a small value, then the absolute errors incurred by the reduction could be so small that they would not jeopardize (d, ϵ_r, δ) -approximation. In particular, as we show in Section 4.3.5, if we reduce every residue $\mathbf{r}_s^{(k)}[v]$ by $\beta_k \cdot \epsilon_r \delta \cdot d(v)$ ($k = 0, 1, \dots, K$ and $v \in V$), then the absolute error in $\frac{\hat{\rho}_s[v]}{d(v)}$ incurred by the residue reduction is at most $\epsilon_r \delta \cdot \sum_{k=0}^K \beta_k$. In other words, if we choose β_k such that $\sum_{k=0}^K \beta_k = 1$, then the resulting absolute error in $\frac{\hat{\rho}_s[v]}{d(v)}$ is at most $\epsilon_r \delta$, which is permissible under (d, ϵ_r, δ) -approximation. The following example demonstrates the benefit of this residue reduction method.

Example 4.1. Suppose that given a graph G , seed node s , $K = 4$, and $\epsilon_r \cdot \delta = 0.01$, HK-Push+ returns residue vectors $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(4)}$ that have the characteristics in Table 4.2. Note that v_1 has the largest 3-hop residue and degree-normalized 3-hop residue, while v_2 has the largest 4-hop residue and degree-normalized 4-hop residue. In addition, all of the 3-hop residues except v_1 's are less than 0.0025, and all of the 4-hop residues except v_2 's are less than 0.075. We can observe that

$$\sum_{\ell=0}^4 \max_{v \in V} \left\{ \frac{\mathbf{r}_s^{(\ell)}[v]}{d(v)} \right\} = 0.0025 + 0.0076 = 0.0101,$$

which is slightly larger than $\epsilon_r \cdot \delta$. In this case, according to Lines 7-8 in TEA, we need to perform $\alpha \cdot \omega$ random walks, where

$$\alpha = \sum_{k=0}^K \sum_{u \in V} \mathbf{r}_s^{(k)}[u] = 0.4.$$

Now suppose that we apply the residue reduction method, setting $\beta_3 = 1/4$ and $\beta_4 = 3/4$. In that case, we reduce every residue $\mathbf{r}_s^{(3)}[v]$ by

$$\beta_3 \cdot \epsilon_r \delta \cdot d(v) = 0.0025 \cdot d(v),$$

and every residue $\mathbf{r}_s^{(4)}[v]$ by

$$\beta_4 \cdot \epsilon_r \delta \cdot d(v) = 0.0075 \cdot d(v).$$

Then, all residues in $\mathbf{r}_s^{(3)}$ and $\mathbf{r}_s^{(4)}$ are reduced to 0, except that $\mathbf{r}_s^{(4)}[v_2]$ is decreased to $0.076 - \beta_4 \cdot \epsilon_r \delta \cdot d(v_2) = 0.001$. Accordingly, $\alpha = \sum_{\ell=0}^4 \max_{v \in V} \mathbf{r}_s^{(\ell)}[v]$ is reduced from 0.4 to 0.001, which implies that the number of random walks required is reduced by 400 times. $\square$

4.3.3 Details of TEA+

Based on the ideas described in Sections 4.3.1 and 4.3.2, we present TEA+, which utilizes HK-Push+ and random walks to compute a (d, ϵ_r, δ) -approximate HKPR vector $\hat{\boldsymbol{\rho}}_s$ with at least $1 - p_f$ probability for any given seed node s . Algorithm 5 illustrates the pseudocode of TEA+. The input parameters of TEA+ are identical to those of TEA, except that TEA+ takes an additional parameter c , which, as mentioned in Section 4.3.1, is used to decide the maximum number K of hops used in HK-Push+.

TEA+ starts by invoking HK-Push+ with the following parameters (Lines 5-6): G , ϵ_r , δ , $K = c \cdot \frac{\log(\frac{1}{\epsilon_r \cdot \delta})}{\log(d)}$, and $n_p = \frac{\omega \cdot t}{2}$, where

$$\omega = \frac{8(1 + \epsilon_r/6) \log(1/p'_f)}{\epsilon_r^2 \cdot \delta},$$

and p'_f is as defined in Equation (4.5) and is pre-computed when G is loaded. Then, HK-Push+ returns an approximate HKPR vector $\hat{\boldsymbol{\rho}}_s$ and $K+1$ residue vectors $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$.

If $\sum_{k=0}^K \max_{u \in V} \left\{ \frac{\mathbf{r}_s^{(k)}[u]}{d(u)} \right\} \leq \epsilon_r \cdot \delta$, then by Theorem 4.2, $\hat{\rho}_s$ is a (d, ϵ_r, δ) -approximate HKPR vector. In that case, TEA+ immediately terminates and returns $\hat{\rho}_s$ (Line 7). Otherwise, TEA+ proceeds to refine $\hat{\rho}_s$ using k -RandomWalk (Lines 8-20). But before that, TEA+ first applies the residue reduction method discussed in Section 4.3.2. Specifically, it decreases each residue value $\mathbf{r}_s^{(k)}[u]$ by $\beta_k \cdot \epsilon_r \delta \cdot d(u)$ (Lines 8-11), where

$$\beta_k = \frac{\sum_{u \in V} \mathbf{r}_s^{(k)}[u]}{\sum_{\ell=0}^K \sum_{u \in V} \mathbf{r}_s^{(\ell)}[u]}.$$

The rationale of this choice of β_k is as follows. First, $\sum_{k=0}^K \beta_k = 1$, which is crucial for (d, ϵ_r, δ) -approximation, as we mention in Section 4.3.2. Second, we set β_k to be proportional to $\sum_{u \in V} \mathbf{r}_s^{(k)}[u]$ because, intuitively, when $\sum_{u \in V} \mathbf{r}_s^{(k)}[u]$ is large, the residue values in $\mathbf{r}_s^{(k)}$ also tend to be large, in which case we need a larger reduction of the residues in $\mathbf{r}_s^{(k)}$ to effectively reduce the number of random walks needed.

After reducing the residues in $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$, TEA+ performs random walks according to the reduced residues, in the same way as TEA does (Lines 12-17). This results in a refined approximate HKPR vector $\hat{\rho}_s$. Then, TEA+ gives $\hat{\rho}_s[v]$ a final touch by adding $\frac{\epsilon_r \cdot \delta}{2} \cdot d(v)$ to each $\hat{\rho}_s[v]$ (Lines 18-19). The intuition of adding this offset to each $\hat{\rho}_s[v]$ is as follows. The residue reduction method leads to an underestimation of each HKPR value, and amount of underestimation is in $[0, \epsilon_r \cdot \delta \cdot d(v)]$. By adding an offset $\frac{\epsilon_r \cdot \delta}{2} \cdot d(v)$ to $\hat{\rho}_s[v]$, the range of the error in $\hat{\rho}_s[v]$ becomes $[-\frac{\epsilon_r \cdot \delta}{2} \cdot d(v), \frac{\epsilon_r \cdot \delta}{2} \cdot d(v)]$, in which case the maximum absolute error in $\hat{\rho}_s[v]$ is reduced by half, which help tightening the accuracy bound of TEA+.

Note that Lines 18-19 in TEA+ can be performed in $O(1)$ time, as we can keep each $\hat{\rho}_s[v]$ unchanged but record the value of $\frac{\epsilon_r \cdot \delta}{2}$ along with $\hat{\rho}_s$. Then, whenever $\hat{\rho}_s[v]$ is accessed, we can add $\frac{\epsilon_r \cdot \delta}{2} \cdot d(v)$ to $\hat{\rho}_s[v]$ on the fly. In addition, for the purpose of local clustering, we can simply ignore this offset of $\frac{\epsilon_r \cdot \delta}{2} \cdot d(v)$ since it does not affect the ranking of nodes based on $\frac{\hat{\rho}_s[v]}{d(v)}$.

4.3.4 Example for TEA+

We use the graph G' in Figure 4.1 to illustrate TEA+. For ease of presentation, we set $t = 3$, $p_f = 10^{-2}$, $\epsilon_r = 0.5$, $\delta = \frac{2\tau}{9}$, and $\tau = 1 - \frac{4}{e^3} \approx 0.8$, and we set c to a constant such

Algorithm 5: TEA+

Input: Graph G , seed node s , constant c , thresholds ϵ_r and δ , and failure probability

p_f

Output: A (d, ϵ_r, δ) -approximate HKPR vector $\hat{\rho}_s$

Lines 1-4 are the same as Lines 1-4 in Algorithm 3;

5 $\omega \leftarrow \frac{8(1+\epsilon_r/6) \log(1/p_f')}{\epsilon_r^2 \delta}, n_p \leftarrow \frac{\omega \cdot t}{2}, K \leftarrow c \cdot \frac{\log(\frac{1}{\epsilon_r \delta})}{\log(d)}$;

6 $(\hat{\rho}_s, \mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}) \leftarrow \text{HK-Push+}(s, \epsilon_r, \delta, K, n_p)$;

7 **if** $\sum_{k=0}^K \max_{u \in V} \left\{ \frac{\mathbf{r}_s^{(k)}[u]}{d(u)} \right\} \leq \epsilon_r \cdot \delta$ **then return** $\hat{\rho}_s$;

8 **for** $k = 0$ **to** K **do**

9 $\beta_k \leftarrow \frac{\sum_{u \in V} \mathbf{r}_s^{(k)}[u]}{\sum_{\ell=0}^K \sum_{u \in V} \mathbf{r}_s^{(\ell)}[u]}$;

10 **for any node** u **with** $\mathbf{r}_s^{(k)}[u] > 0$ **do**

11 $\mathbf{r}_s^{(k)}[u] = \max \left\{ 0, \mathbf{r}_s^{(k)}[u] - \beta_k \cdot \epsilon_r \delta \cdot d(u) \right\}$;

Lines 12-17 are the same as Lines 7-12 in Algorithm 3;

18 **for** $v \in V$ **do**

19 $\hat{\rho}_s[v] \leftarrow \hat{\rho}_s[v] + \frac{\epsilon_r \delta}{2} \cdot d(v)$;

20 **return** $\hat{\rho}_s$;

Figure 4.1: A graph G' for illustrating TEA+.

Table 4.3: The reserves and residues after the first round of push operations.

	s	v_1	v_2	v_3	v_4	v_5	v_6	v_7
$\mathbf{q}_s$	$\frac{1}{e^3}$	0	0	0	0	0	0	0
$\mathbf{r}_s^{(1)}$	0	$\frac{e^3-1}{2e^3}$	$\frac{e^3-1}{2e^3}$	0	0	0	0	0

that $K = 2$. Consider that we invoke Algorithm 5 on G' with seed node s and above parameters as inputs. TEA+ first computes $\omega \approx \frac{970}{\tau}$ and $n_p \approx \frac{1455}{\tau}$, and then invokes Algorithm 4 with seed node s and parameters ϵ_r , δ , K , and n_p as inputs.

Initially, Algorithm 4 sets $\mathbf{r}_s^{(0)}[s] = 1$ and $k = 0$. In that case, $k < K$, and $\frac{\mathbf{r}_s^{(0)}[s]}{d(s)} =$

Table 4.4: The reserves and residues after the second round of push operations.

	s	v_1	v_2	v_3	v_4	v_5	v_6	v_7
$\mathbf{q}_s$	$\frac{1}{e^3}$	$\frac{3}{2e^3}$	0	0	0	0	0	0
$\mathbf{r}_s^{(1)}$	0	0	$\frac{e^3-1}{2e^3}$	0	0	0	0	0
$\mathbf{r}_s^{(2)}$	$\frac{\tau}{6}$	0	$\frac{\tau}{6}$	$\frac{\tau}{6}$	0	0	0	0

Table 4.5: The reserves and residues after the third round of push operations.

	s	v_1	v_2	v_3	v_4	v_5	v_6	v_7
$\mathbf{q}_s$	$\frac{1}{e^3}$	$\frac{3}{2e^3}$	$\frac{3}{2e^3}$	0	0	0	0	0
$\mathbf{r}_s^{(1)}$	0	0	0	0	0	0	0	0
$\mathbf{r}_s^{(2)}$	$\frac{\tau}{4}$	$\frac{\tau}{12}$	$\frac{\tau}{6}$	$\frac{\tau}{6}$	$\frac{\tau}{12}$	$\frac{\tau}{12}$	$\frac{\tau}{12}$	$\frac{\tau}{12}$

$0.5 > \frac{\epsilon_r \delta}{K} \approx 0.0445$. Accordingly, Algorithm 4 performs push operations from s . It first converts $\frac{\eta(0)}{\psi(0)} = \frac{1}{e^3}$ portion of s 's reserve $\mathbf{r}_s^{(0)}[s]$ into its reserve $\mathbf{q}_s[s]$, and then distributes $\frac{1}{2}(1 - \frac{\eta(0)}{\psi(0)}) = \frac{e^3-1}{2e^3}$ portion to each of its two neighbors, i.e., v_1 and v_2 . Table 4.3 shows the reserves and residues after the first round of push operations.

As shown in Table 4.3 and Figure 4.1, $\frac{\mathbf{r}_s^{(1)}[v_1]}{d(v_1)} \approx 0.1584$ and $\frac{\mathbf{r}_s^{(1)}[v_2]}{d(v_2)} \approx 0.0792$, both of which are greater than $\frac{\epsilon_r \delta}{K} \approx 0.0445$. In addition, $k = 1 < K$, and the number of push operations performed i is 2, which is smaller than n_p . Hence, Algorithm 4 starts the second round of push operations on $\mathbf{r}_s^{(1)}[v_1]$. In that case, it converts $\frac{\eta(1)}{\psi(1)}$ portion of $\mathbf{r}_s^{(1)}[v_1]$ into v_1 's reserve, and distributes $\frac{1}{3}(1 - \frac{\eta(1)}{\psi(1)}) \cdot \mathbf{r}_s^{(1)}[v_1] = \frac{\tau}{6}$ residue to each of v_1 's three neighbors s, v_2 and v_3 . Table 4.4 lists the reserves and residues after this round of push operations.

In the third round, Algorithm 4 converts $\frac{\eta(1)}{\psi(1)}$ portion of $\mathbf{r}_s^{(1)}[v_2]$ into the reserve of v_2 . Meanwhile, each of v_2 's neighbors receives $\frac{1}{6}(1 - \frac{\eta(1)}{\psi(1)}) \cdot \mathbf{r}_s^{(1)}[v_2]$ residue. After this round, $k = K = 2$, and hence, Algorithm 4 terminates with the reserves in Table 4.5 as the approximate HKPR values in $\hat{\rho}_s$.

Based on Table 4.4 and Figure 4.1, we have the following results:

$$\frac{\mathbf{r}_s^{(2)}[s]}{d(s)} = \frac{\tau}{8}, \frac{\mathbf{r}_s^{(2)}[v_1]}{d(v_1)} = \frac{\tau}{36}, \frac{\mathbf{r}_s^{(2)}[v_2]}{d(v_2)} = \frac{\tau}{36}, \frac{\mathbf{r}_s^{(2)}[v_3]}{d(v_3)} = \frac{\tau}{6},$$

$$\frac{\mathbf{r}_s^{(2)}[v_4]}{d(v_4)} = \frac{\mathbf{r}_s^{(2)}[v_5]}{d(v_5)} = \frac{\mathbf{r}_s^{(2)}[v_6]}{d(v_6)} = \frac{\mathbf{r}_s^{(2)}[v_6]}{d(v_6)} = \frac{\tau}{12}.$$

Since $\sum_{k=0}^2 \max_{u \in V} \left\{ \frac{\mathbf{r}_s^{(k)}[u]}{d(u)} \right\} = \frac{\tau}{6} > \epsilon_r \cdot \delta = \frac{\tau}{9}$, Algorithm 5 does not terminate, but would perform random walks to refine $\hat{\boldsymbol{\rho}}_s$. It first computes $\beta_0 = \beta_1 = 0$ and $\beta_2 = 1$, and then reduces the residue of each node v by $\mathbf{r}_s^{(2)}[v] = \max \left\{ 0, \mathbf{r}_s^{(2)}[v] - \beta_2 \cdot \epsilon_r \delta \cdot d(v) \right\}$. This leads to

$$\begin{aligned} \mathbf{r}_s^{(2)}[v_3] &= \mathbf{r}_s^{(2)}[v_3] - \epsilon_r \cdot \delta = \frac{\tau}{18}, \\ \mathbf{r}_s^{(2)}[s] &= \mathbf{r}_s^{(2)}[s] - 2 \cdot \epsilon_r \delta = \frac{\tau}{36}, \end{aligned}$$

and other residues become 0. The number of random walks required by Algorithm 5 is therefore

$$n_r = \alpha \cdot \omega = \sum_{k=0}^2 \sum_{u \in V} \mathbf{r}_s^{(k)}[u] \cdot \omega = \frac{\tau}{12} \cdot \omega \approx 81.$$

Before performing random walks from s and v_3 , Algorithm 5 first builds an alias structure on all nodes, such that entries $(s, 2)$ and $(v_3, 2)$ would be sampled with probabilities $\frac{\mathbf{r}_s^{(2)}[s]}{\alpha} = \frac{1}{3}$ and $\frac{\mathbf{r}_s^{(2)}[v_3]}{\alpha} = \frac{2}{3}$, respectively. After that, it invokes Algorithm 2 for $n_r = 81$ times. In each invocation, it samples the starting entry $(v, 2)$ from this alias structure, and at the ℓ -th step, it terminates at the current node u with probability $\frac{\eta(2+\ell)}{\psi(2+\ell)}$, and traverses to a randomly selected neighbor of u with the other $1 - \frac{\eta(2+\ell)}{\psi(2+\ell)}$ probability. Once it terminates at a node w , Algorithm 5 increases w 's approximate HKPR value $\hat{\boldsymbol{\rho}}_s[w]$ by $\frac{\alpha}{n_r} = \frac{\tau}{970}$. TEA+ returns $\hat{\boldsymbol{\rho}}_s$ as an approximation of the exact HKPR vector $\boldsymbol{\rho}_s$ after 81 random walks.

Finally, we perform a sweep over $\hat{\boldsymbol{\rho}}_s$ to find a cut with the best conductance. That is, we sort the nodes v by $\frac{\hat{\boldsymbol{\rho}}_s[v]}{d(v)}$ in descending order. If the sorted order is $s, v_1, v_3, v_2, v_4 - v_7$, then the sweep would produce $\{s, v_1, v_3\}$ as the best cut, as its conductance $\frac{1}{3}$ is smaller than other cuts encountered during the sweep.

4.3.5 Analysis

Correctness. Let $\mathbf{q}_s$ denote the approximate HKPR vector returned by HK-Push+ in Line 6 of TEA+, $\mathbf{r}_s^{(0)}, \dots, \mathbf{r}_s^{(K)}$ be the residue vectors output by HK-Push+ at the same

step, and $\widehat{\rho}_s$ be the final version of the HKPR vector output by TEA+. We define a residue vector $\mathbf{rb}_s^{(k)}$ as:

$$\mathbf{rb}_s^{(k)}[u] = \min \{ \mathbf{r}_s^{(k)}[u], \beta_k \cdot \epsilon_r \delta \cdot d(u) \}. \quad (4.14)$$

Observe that $\mathbf{rb}_s^{(k)}[u]$ equals the amount of residue reduction on $\mathbf{r}_s^{(k)}[u]$ occurred in Lines 8-11 of TEA+.

Similar to the correctness analysis in Section 4.2.3, by Lemma 4.1, we have the following equation for any node v :

$$\rho_s[v] = \mathbf{q}_s[v] + \mathbf{a}_s[v] + \mathbf{b}_s[v], \quad (4.15)$$

where

$$\mathbf{a}_s[v] = \alpha \cdot \sum_{k=0}^K \sum_{u \in V} \frac{\mathbf{r}_s^{(k)}[u]}{\alpha} \cdot \mathbf{h}_u^{(k)}[v], \text{ and} \quad (4.16)$$

$$\mathbf{b}_s[v] = \sum_{k=0}^K \sum_{u \in V} \mathbf{rb}_s^{(k)}[u] \cdot \mathbf{h}_u^{(k)}[v]. \quad (4.17)$$

Then, the approximation error in each $\widehat{\rho}_s[v]$ can be regarded as the sum of two approximation errors for $\mathbf{a}_s[v]$ and $\mathbf{b}_s[v]$, respectively. The error in $\mathbf{a}_s[v]$ is caused by sampling errors in k -RandomWalk, and hence, it can be bounded using the Chernoff bound, in a way similar to the analysis in Section 4.2.3. Meanwhile, the error in $\mathbf{b}_s[v]$ is due to the residue reduction procedure in Lines 8-11 of TEA+. In what follows, we present an analysis of the error in $\mathbf{b}_s[v]$.

By Equation (4.14), for any node $u \in V$ and $k \in [0, K]$, the amount of residue reduction on $\mathbf{r}_s^{(k)}[u]$ satisfies $0 \leq \mathbf{rb}_s^{(k)}[u] \leq \beta_k \cdot \epsilon_r \delta \cdot d(u)$. Combining this with Equation (4.17), we have

$$0 \leq \mathbf{b}_s[v] \leq \sum_{k=0}^K \left(\beta_k \cdot \epsilon_r \delta \sum_{u \in V} d(u) \cdot \mathbf{h}_u^{(k)}[v] \right). \quad (4.18)$$

Lemma 4.6 ([113]). *Let u and v be any two nodes in G , and $\mathbf{P}^k[u, v]$ (resp. $\mathbf{P}^k[v, u]$) be the probability that a length- k random walk from u ends at v (resp. from v ends at u). Then,*

$$\frac{\mathbf{P}^k[u, v]}{d(v)} = \frac{\mathbf{P}^k[v, u]}{d(u)}. \quad \square$$

By Lemma 4.6 and the definition of $\mathbf{h}_u^{(k)}[v]$ in Equation (4.3), for any node $v \in V$ and $k \in [0, K]$, we have

$$\begin{aligned} \sum_{u \in V} d(u) \cdot \mathbf{h}_u^{(k)}[v] &= d(v) \cdot \sum_{u \in V} \mathbf{h}_v^{(k)}[u] \\ &= d(v) \cdot \sum_{\ell=0}^{\infty} \left[\frac{\eta(k+\ell)}{\psi(k)} \cdot \sum_{u \in V} \mathbf{P}^\ell[v, u] \right] \\ &= d(v) \cdot \sum_{\ell=0}^{\infty} \frac{\eta(k+\ell)}{\psi(k)} = d(v). \end{aligned} \quad (4.19)$$

Combining Equations (4.18) and (4.19), we have

$$0 \leq \mathbf{b}_s[v] \leq d(v) \cdot \epsilon_r \delta. \quad (4.20)$$

Therefore, estimating $\mathbf{b}_s[v]$ as $\frac{\epsilon_r \delta}{2} \cdot d(v)$ incurs an absolute error of at most $\frac{\epsilon_r \delta}{2} \cdot d(v)$.

Based on the above analysis, we establish the accuracy guarantee of TEA+ as follows:

Theorem 4.3. *TEA+ outputs a (d, ϵ_r, δ) -approximate HKPR vector $\hat{\boldsymbol{\rho}}_s$ with probability at least $1 - p_f$.*

Proof. First, consider the case where TEA+ terminates at Line 7, i.e., $\sum_{k=0}^K \max_{u \in V} \left\{ \frac{\mathbf{r}_s^{(k)}[u]}{d(u)} \right\} \leq \epsilon_a$. In that case, by Theorem 4.2, for any node $v \in V$, $\left| \frac{\hat{\boldsymbol{\rho}}_s[v]}{d(v)} - \frac{\boldsymbol{\rho}_s[v]}{d(v)} \right| \leq \epsilon_a = \epsilon_r \cdot \delta$. This indicates that, for any node v with $\boldsymbol{\rho}_s[v] > d(v) \cdot \delta$, $\left| \frac{\hat{\boldsymbol{\rho}}_s[v]}{d(v)} - \frac{\boldsymbol{\rho}_s[v]}{d(v)} \right| \leq \epsilon_r \cdot \frac{\boldsymbol{\rho}_s[v]}{d(v)}$. In addition, for any node v with $\boldsymbol{\rho}_s[v] \leq d(v) \cdot \delta$, $\left| \frac{\hat{\boldsymbol{\rho}}_s[v]}{d(v)} - \frac{\boldsymbol{\rho}_s[v]}{d(v)} \right| \leq \epsilon_r \cdot \delta$. Thus, $\hat{\boldsymbol{\rho}}_s$ is a (d, ϵ_r, δ) -approximate HKPR vector.

Now consider the case where TEA+ does not terminate at Line 7. Let Y_i be as defined in the context of Equation (4.9), and let $Y = \sum_{i=1}^{n_r} Y_i$. By Lines 17 and 19 of TEA+,

$$\hat{\boldsymbol{\rho}}_s[v] = \mathbf{q}_s[v] + \frac{Y \cdot \alpha}{n_r} + \frac{\epsilon_r \delta}{2} \cdot d(v).$$

By Equation (4.10), the expectation of Y is

$$\mathbb{E}[Y] = \mathbb{E}\left[\sum_{i=1}^{n_r} Y_i\right] = \frac{n_r}{\alpha} (\boldsymbol{\rho}_s[v] - \mathbf{q}_s[v] - \mathbf{b}_s[v]) \leq \frac{n_r}{\alpha} \cdot \boldsymbol{\rho}_s[v].$$

Let $\lambda = \frac{n_r \epsilon_r}{2\alpha} \cdot \rho_s[v]$ and n_r be as defined in Line 13 of TEA+. By the Chernoff bound (see Lemma 4.5), for any node v in V with $\rho_s[v] > d(v) \cdot \delta$, we have

$$\begin{aligned} & \mathbb{P}[|Y - \mathbb{E}[Y]| \geq \lambda] \\ &= \mathbb{P}\left[\left|\widehat{\rho}_s[v] - \rho_s[v] + \mathbf{b}_s[v] - \frac{\epsilon_r \delta}{2} \cdot d(v)\right| \geq \frac{\epsilon_r}{2} \cdot \rho_s[v]\right] \\ &\leq \exp\left(-\frac{n_r \cdot \epsilon_r^2 \cdot \rho_s[v]}{8\alpha \cdot (1 + \epsilon_r/6)}\right) \leq (p'_f)^{d(v)}. \end{aligned}$$

On the other hand, let $\lambda = \frac{n_r \epsilon_r \delta d(v)}{2\alpha}$. Then, for any node v in V with $\rho_s[v] \leq d(v) \cdot \delta$, we have

$$\begin{aligned} & \mathbb{P}[|Y - \mathbb{E}[Y]| \geq \lambda] \\ &= \mathbb{P}\left[\left|\widehat{\rho}_s[v] - \rho_s[v] + \mathbf{b}_s[v] - \frac{\epsilon_r \delta}{2} \cdot d(v)\right| \geq \frac{\epsilon_r \delta}{2} \cdot d(v)\right] \\ &\leq \exp\left(-\frac{n_r \cdot \epsilon_r^2 \delta^2 d^2(v)}{8\alpha(1 + \epsilon_r/6) \cdot \rho_s[v]}\right) \leq (p'_f)^{d(v)}. \end{aligned}$$

By union bound, for $V_1 = \{v | v \in V \text{ s.t. } \rho_s[v] > d(v) \cdot \delta\}$, we have

$$\begin{aligned} & \mathbb{P}\left[\bigcup_{v \in V_1} \left\{\left|\frac{\widehat{\rho}_s[v] - \rho_s[v] + \mathbf{b}_s[v]}{d(v)} - \frac{\epsilon_r \delta}{2}\right| \geq \frac{\epsilon_r}{2} \cdot \frac{\rho_s[v]}{d(v)}\right\}\right] \\ &\leq \sum_{v \in V_1} (p'_f)^{d(v)}, \end{aligned}$$

and for $V_2 = \{v | v \in V \text{ s.t. } \rho_s[v] \leq d(v) \cdot \delta\}$, we have

$$\mathbb{P}\left[\bigcup_{v \in V_2} \left\{\left|\frac{\widehat{\rho}_s[v] - \rho_s[v] + \mathbf{b}_s[v]}{d(v)} - \frac{\epsilon_r \delta}{2}\right| \geq \frac{\epsilon_r \delta}{2}\right\}\right] \leq \sum_{v \in V_2} (p'_f)^{d(v)}.$$

By Inequality (4.20), $\left|\frac{\mathbf{b}_s[v]}{d(v)} - \frac{\epsilon_r \delta}{2}\right| \leq \frac{\epsilon_r \delta}{2}$. Then, we have the following results, respectively. With probability at least $1 - \sum_{v \in V} (p'_f)^{d(v)}$, for every node v in V with $\rho_s[v] > d(v) \cdot \delta$,

$$\left|\frac{\widehat{\rho}_s[v]}{d(v)} - \frac{\rho_s[v]}{d(v)}\right| \leq \frac{\epsilon_r}{2} \cdot \frac{\rho_s[v]}{d(v)} + \frac{\epsilon_r \delta}{2} \leq \epsilon_r \cdot \frac{\rho_s[v]}{d(v)},$$

and for every node v in V with $\rho_s[v] \leq d(v) \cdot \delta$,

$$\left|\frac{\widehat{\rho}_s[v]}{d(v)} - \frac{\rho_s[v]}{d(v)}\right| \leq \epsilon_r \delta.$$

By the definition of p'_f in Equation (4.5), the total failure probability will be at most $\sum_{v \in V} (p'_f)^{d(v)} \leq p_f$. Therefore, $\hat{\rho}_s$ is a (d, ϵ_r, δ) -approximate HKPR vector with probability at least $1 - p_f$. $\square$

Time and Space Complexities. The time and space complexities of TEA+ depend on its two main components: HK-Push+ and k -RandomWalk. As discussed in Section 4.3.1, both the computation and space overheads of HK-Push+ are $O(n_p)$. Since TEA+ sets $n_p = \frac{\omega \cdot t}{2}$, its invocation of HK-Push+ incurs $O\left(\frac{t \cdot \log(n/p_f)}{\epsilon_r^2 \cdot \delta}\right)$ time and space costs. Meanwhile, the total number of random walks conducted by TEA+ is no more than that by TEA, and hence, the computational and space costs of generating random walks in TEA+ are both $O\left(\frac{t \cdot \log(n/p_f)}{\epsilon_r^2 \cdot \delta}\right)$ in expectation. Thus, the expected time and space complexities of TEA+ are $O\left(\frac{t \cdot \log(n/p_f)}{\epsilon_r^2 \cdot \delta}\right)$ and $O\left(m + n + \frac{t \cdot \log(n/p_f)}{\epsilon_r^2 \cdot \delta}\right)$, respectively, where the $m + n$ term is due to the space required by the input graph.

4.4 Experiments

4.4.1 Experimental Setup

We conduct all experiments on a Linux server with a Intel Xeon(R) E5-2650 v2@2.60GHz CPU and 64GB RAM. For fair comparison, all algorithms are implemented in C++ and compiled by g++ 4.8 with -O3 optimization.

We use 6 undirected real-world graphs and 2 synthetic graphs which are used in recent work [22, 36, 41] as benchmark datasets (Table 4.6). We obtain *DBLP*, *Youtube*, *Orkut*, *LiveJournal*, and *Friendster* from [92]. *PLC* is a synthetic graph, and it is generated by Holme and Kim algorithm for generating graphs with powerlaw degree distribution and approximate average clustering. *3D-grid* is a synthetic grid graph in 3-dimensional space where every node has six edges, each connecting it to its 2 neighbors in each dimension. *Twitter* is a symmetrized version of a snapshot of the Twitter network [37]. For each dataset, we pick 50 seed nodes uniformly at random as our query sets.

Unless specified otherwise, following previous work [9, 36], we set heat constant $t = 5$. In addition, for all randomized algorithms, we set failure probability $p_f = 10^{-6}$. We report the average query time (measured in wall-clock time) of each algorithm on each dataset with various parameter settings. Note that the y -axis is in log-scale and the measurement unit for running time is millisecond (ms).

Table 4.6: Statistics of graph datasets.

Dataset	n	m	$\bar{d}$
<i>DBLP</i>	317,080	1,049,866	6.62
<i>Youtube</i>	1,134,890	2,987,624	5.27
<i>PLC</i>	2,000,000	9,999,961	9.99
<i>Orkut</i>	3,072,441	117,185,083	76.28
<i>LiveJournal</i>	3,997,962	34,681,189	17.35
<i>3D-grid</i>	9,938,375	29,676,450	5.97
<i>Twitter</i>	41,652,231	1,202,513,046	57.74
<i>Friendster</i>	65,608,366	1,806,067,135	55.06

4.4.2 Tuning Parameter c for TEA+

First, we experimentally study how to set the parameter c so as to obtain the best performance for TEA+ in practice. We run TEA+ with parameters $\epsilon_r = 0.5$, $\delta = \frac{1}{n}$, and varying c from 0.5 to 5 on all 8 graphs.

Figure 4.2 plots the running time of TEA+ on each dataset for different c . We omit the results for *Twitter* and *Friendster* when $c = 0.5$, because it takes several hours to finish one query. We can make the following observations. For each dataset, the running time decreases first as c grows. The reason is that TEA+ degrades to Monte-Carlo when c is very small, and if we keep increasing c , HK-Push+ will perform more push operations so as to reduce the number of random walks until c balances the costs incurred for HK-Push+ and k -RandomWalk. On the other hand, when c increases further, the overhead of HK-Push+ goes up gradually, disrupting the balance between HK-Push+ and k -RandomWalk. This leads to higher running time. More specifically, we can see that for graphs with small average degree including *DBLP*, *Youtube*, *PLC* and *3D-grid*, the costs are minimized when c is around 2. On the other hand, for graphs with high average degree (*e.g.*, *Orkut*, *LiveJournal*, *Twitter*, and *Friendster*), we note that $c = 2.5$ achieves the best performance. Based on the above observations, a good value choice for c is 2.5, when the overheads on most of the graphs are minimized. In the sequel, we set $c = 2.5$.

 Figure 4.2: Running time of TEA+ vs c (best viewed in color).

 Figure 4.3: Running time vs ϵ_r .

4.4.3 Comparison between TEA and TEA+

In this set of experiments, we compare TEA+ with TEA based on identical theoretical accuracy guarantees. For both TEA and TEA+, we set the relative error threshold $\epsilon_r = 0.5$. Since the best values for r_{max} vary largely for different parameter settings and various datasets, we are unable to find a universal optimal value for r_{max} . Instead, we tune r_{max} for TEA with different error thresholds on each dataset separately. That is, we scale $\frac{1}{\omega \cdot t}$ up or down such that the costs for HK-Push and k -RandomWalk in TEA are roughly balanced and the total cost is minimized.

Figure 4.3 reports the computational time of TEA and TEA+ on all datasets when varying ϵ_r from 0.1 to 0.9 and fixing $\delta = 10^{-6}$. Observe that TEA+ always outperforms

Figure 4.4: Running time vs conductance for local clustering queries (best viewed in color).

TEA markedly for all datasets. In particular, when the relative error threshold ϵ_r is large (e.g., 0.5 – 0.9), TEA+ is one to two orders of magnitude faster than TEA on most of datasets, but only around 5 times faster on *3D-grid*. This is caused by the fact that each node in *3D-grid* has six neighbors, thus residues will drop below the threshold quickly and both TEA and TEA+ require very few random walks. As we keep decreasing ϵ_r , the gap between TEA and TEA+ is narrowed. Especially, when $\epsilon_r = 0.1$, TEA+ achieves around $5\times$ to $10\times$ speedup. The reasons are as follows. When the relative error thresholds are large, TEA+ only needs a small number of push operations and random walks due to the new termination conditions of HK-Push+ and residue reduction method compared to TEA. However, when the relative error thresholds are very small, the termination conditions of HK-Push+ are harder to satisfy and as a result incurs much higher costs to terminate. Furthermore, the residue reduction method is not able to reduce the number of random walks significantly since the residue sum is already small. Thus, both TEA and TEA+ perform many push operations and random walks, and as a result, the speedup is modest. The results demonstrate the power of new termination conditions of HK-Push+ and residue reduction method in TEA+, especially when the error thresholds are not very small. *In summary, TEA+ outperforms TEA without sacrificing theoretical accuracies of HKPR values.*

4.4.4 Comparisons with Competitors

We compare TEA and TEA+ against ClusterHKPR, SimpleLocal [137], CRD [46], Monte-Carlo and HK-Relax in terms of clustering quality and efficiency (or memory overheads). Recall that Monte-Carlo is a random-walk-based approach as described in Section 4.1.2. Monte-Carlo accepts two thresholds ϵ_r and δ , and a failure probability p_f as inputs. It performs $\frac{2(1+\epsilon_r/3)\log(n/p_f)}{\epsilon_r^2\delta}$ random walks from the seed node s and returns a (d, ϵ_r, δ) -approximate HKPR vector for s with probability at least $1 - p_f$. HK-Relax ensures ϵ_a absolute error in each $\frac{\hat{p}_s[v]}{d(v)}$, which is incomparable with the accuracy guarantees of other three methods, that is (d, ϵ_r, δ) -approximation guarantees. Hence, we do not compare all algorithms under the same theoretical accuracy guarantees. Instead, we evaluate each method by its empirical clustering quality (*i.e.*, conductance) and empirical running time (or memory overheads) with various parameter settings and find the method that achieves the best trade-off in terms of clustering quality and running time (or memory overheads). HK-Relax only has one internal parameter ϵ_a , which is the absolute error threshold. We vary it in $\{10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}\}$ in our experiments. Since Monte-Carlo, TEA, and TEA+ have almost the same parameters, we set relative error threshold $\epsilon_r = 0.5$, and δ is varied in $\{2 \times 10^{-8}, 2 \times 10^{-7}, 2 \times 10^{-6}, 2 \times 10^{-5}, 2 \times 10^{-4}\}$ for all of them. ClusterHKPR has one internal parameter ϵ , which is varied in $\{0.005, 0.01, 0.02, 0.05, 0.1\}$. SimpleLocal has one locality parameter δ . We vary it in $\{0.005, 0.01, 0.02, 0.05, 0.1\}$. In addition, we vary the number of iterations of CRD in $\{7, 10, 15, 20, 30\}$ and keep other parameters default. For fair comparison, we let the x -axis be the average conductance of the output clusters and the y -axis be the average running time (or memory overheads), depicting the empirical clustering quality and empirical efficiency (or memory overheads), respectively.

Figure 4.4 shows the average conductance of the output clusters and the the average running time when varying the aforementioned parameters for each algorithm. We can make the following observations. First, for each algorithm, when the error thresholds (*i.e.*, ϵ_a, ϵ and δ) become smaller or the number of iterations increase, the conductance of the output clusters reduces (*i.e.*, the quality of the output clusters improves), as well as the computational time goes up markedly, which accords with their theoretical time complexities. In particular, SimpleLocal incurs very high running time as well as poor cluster quality due to its high time complexity and the fact that it is mainly devised

Figure 4.5: Memory cost vs. conductance.

Figure 4.6: Running time vs. NDCG for computing normalized HKPR (best viewed in color).

for recovering the cluster for a subset of nodes of the cluster rather than detecting a cluster for a single seed node. We also note that CRD shows a better performance than SimpleLocal. However, it is still much slower than HKPR-based methods. Hence, we omit the results of SimpleLocal on the remaining datasets and that of CRD on *Orkut*, *LiveJournal*, *3D-grid*, *Twitter* and *Friendster*.

Second, we can see that Monte-Carlo and ClusterHKPR take several minutes to finish one local clustering query on all graph datasets in order to find clusters with small conductance. Hence, it is outperformed by HK-Relax, TEA and TEA+ by 1 to 3 orders of magnitude when they output clusters with almost the same conductance. This is due to the fact that Monte-Carlo and ClusterHKPR require performing a large number of random walks. In fact, this is consistent with the experimental results in prior work [22, 41].

Table 4.7: The result of evaluating all algorithms on finding real-world communities.

Data	F_1 -measure				
	ClusterHKPR	Monte-Carlo	HK-Relax	TEA	TEA+
<i>DBLP</i>	0.13655	0.13631	0.13592	0.13679	0.136699
<i>Youtube</i>	0.10113	0.10097	0.09858	0.10133	0.10334
<i>LiveJournal</i>	0.64644	0.65105	0.64516	0.64959	0.67
<i>Orkut</i>	0.18497	0.18464	0.19375	0.19333	0.19636
Data	Running Time (ms)				
	ClusterHKPR	Monte-Carlo	HK-Relax	TEA	TEA+
<i>DBLP</i>	3053.95	2891.64	297.78	176	109.66
<i>Youtube</i>	7.76	7.2	8.11	2.49	2
<i>LiveJournal</i>	1.34665	1.2	3.57	0.55	0.29
<i>Orkut</i>	29.95	29.35	62.17	24.78	14.86

Figure 4.7: Effect of subgraph densities (best viewed in color).

Moreover, it can be observed that **HK-Relax** always runs faster than **Monte-Carlo** and **ClusterHKPR** and achieves more than $10\times$ speedup on all datasets except *Orkut*, *Twitter*, and *Friendster*. To explain this phenomenon, recall that **HK-Relax** requires iteratively pushing residuals to $1 - K$ -hop nodes from the seed node and K is very large (see Chapter 3). However, on graphs with large average degrees (*Orkut*, *Twitter*, and *Friendster*) the number of push operations increases dramatically after several hops from the seed node.

Third, **TEA+** outperforms **HK-Relax** by more than $10\times$ speedup on *PLC*, *Orkut*, *Twitter*, and *Friendster*, and more than $4\times$ speedup on the rest of graphs. The speedup is achieved by new termination conditions for **HK-Push+** and the residue reduction method for reducing the number random walks. However, we note that the speedup on *DBLP*, *Youtube*, *LiveJournal* and *3D-grid* is not as significant as that on *PLC*, *Orkut*, *Twitter*, and *Friendster*. The reason is that these graphs either have large clustering coefficients [165] or small average degrees. The first one can also be observed in our experiments. With the same parameters as inputs to all three algorithms, the conductance values of output clusters from *PLC*, *Orkut*, *Twitter*, and *Friendster* are clearly greater than those from the remaining four graphs (i.e., *DBLP*, *Youtube*, *LiveJournal* and *3D-grid*). This implies that nodes in these four graphs are more likely to cluster together, and then residues on these graphs tend to be propagated within a small cluster of nodes when **HK-Push+** is performed. Now consider the second reason. Recall that **HK-Push+** iteratively distributes residues to neighbors evenly before any termination condition is satisfied. Since the average degrees are small, the residues that each node receives will be large. Additionally, it needs more iterations to distribute the residues to more nodes, which may not be done before termination. As a result of these two factors, a few nodes will hold large residues rather than many nodes holding small residues. Consequently, the residue reduction method in **TEA+** fails to significantly reduce the number of random walks. Note that **HK-Relax**, **TEA** and **TEA+** all terminate very quickly on *3D-grid* (less than 10 milliseconds), which is consistent with the observation in [41]. This is due to the fact that each node in *3D-grid* has six neighbors and the residues will drop below the residue threshold quickly after performing several rounds of push operations.

In addition, we also note that **TEA** fails to achieve considerable speedup compared with **HK-Relax**. Especially on *DBLP*, *Youtube*, *LiveJournal*, and *3D-grid*, **TEA**'s perfor-

Figure 4.8: Effect of heat constant t on *DBLP* (best viewed in color).

mance degrades to the same level as that of HK-Relax. This is also caused by aforementioned high clustering coefficients and small average degrees of these graphs. TEA+ is around $4\times$ faster than TEA on *Orkut*, *Twitter*, and *Friendster*. In summary, our experiments demonstrate the power of new termination condition of HK-Push+ and residue reduction method, which reduce many push operations and random walks without sacrificing the cluster qualities.

Figure 4.5 shows the memory overheads (including the space required to store the input graph) for all datasets by varying the error thresholds. First, we observe that the memory overheads increase with the reduction in error thresholds as more space is required to store residues and HKPR values. In particular, memory overheads on *3D-grid* and *Friendster* remain almost stable for all algorithms. As each node in *3D-grid* is connected to six neighbors, ClusterHKPR, Monte-Carlo, HK-Relax, TEA and TEA+ easily detect the large set of nodes around the seed nodes and the size of memory allocated for storing HKPR values for this set of nodes remains stable. In *Friendster*, the input graph size is too large and memory usage is not impacted remarkably. Second, we observe that the memory overheads of all algorithms are roughly comparable because space usage is dominated by the storage of the input graph, and hence, in terms of memory cost, there is no significant advantage to choose one algorithm over another.

4.4.5 Ranking Accuracy of Normalized HKPR

In this set of experiments, we evaluate the accuracy and efficiency of each method for computing normalized HKPR values (*i.e.*, $\frac{\rho_s[v]}{d(v)}$). First, we randomly select 50 seed nodes and apply the power method [40] with 40 iterations to compute the ground-truth normalized

Figure 4.9: Effect of heat constant t on PLC (best viewed in color).

HKPR values (we omit the large datasets due to time and memory limitations). Following the experimental settings in Section 4.4.1, we run HK-Relax, ClusterHKPR, Monte-Carlo, TEA and TEA+ to generate normalized HKPR values for the selected seed nodes with varied error thresholds. Specifically, we vary ϵ in $\{10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}\}$ for HK-Relax, ϵ in $\{0.01, 0.02, 0.05, 0.1, 0.2, 0.3\}$ for ClusterHKPR, and we set $\epsilon_r = 0.5$ and δ is varied in $\{2 \times 10^{-8}, 2 \times 10^{-7}, 2 \times 10^{-6}, 2 \times 10^{-5}, 2 \times 10^{-4}, 2 \times 10^{-3}\}$ for Monte-Carlo, TEA and TEA+, respectively. Then, we evaluate the accuracy of each method by using *Normalized Discounted Cumulative Gain (NDCG)* [77], which is a classic metric for evaluating ranking results.

Figure 4.6 reports the performance of each method on four datasets. We can make the following observations. First, as we reduce the error thresholds, both the running time and NDCG of each method increase markedly, which is consistent with their theoretical guarantees. Second, TEA+ consistently incurs least running time while achieving the same NDCG compared to the competing methods. In addition, TEA is $2 \times - 8 \times$ slower than TEA+ while HK-Relax runs even more slowly. Especially on PLC and $Orkut$ datasets, HK-Relax's performance degrades to the same level of ClusterHKPR and Monte-Carlo. Although ClusterHKPR and Monte-Carlo also provide relative-error guarantees, they still incur the highest overheads because they require a large number of random walks. Third, we note that the efficiency and ranking accuracy results accord with the efficiency and clustering quality results reported in Section 4.4.4. This demonstrates the relationship between ranking accuracy of normalized HKPR and the quality of HKPR-based clustering algorithms, emphasizing why our methods produce clusters with smaller conductance than the competing ones.

4.4.6 Clusters Produced vs. Ground-truth

We collect the top 5,000 ground-truth communities in *DBLP*, *Youtube*, *LiveJournal* and *Orkut* datasets from [92]. We select 100 seed nodes from 100 known communities of size greater than 100 randomly as the query set. For all algorithms, we vary t from 3 to 10 ($t > 10$ would give us clusters with substantially lower quality) and their error thresholds respectively to produce clusters with highest average F_1 -measure (i.e., harmonic mean of precision and recall). More specifically, we vary ϵ from 0.005 to 0.35 for ClusterHKPR, ϵ from 10^{-8} to 10^{-1} for HK-Relax. In addition, we fix $\epsilon_r = 0.5$ and vary δ from 10^{-8} to 10^{-1} for Monte-Carlo, TEA and TEA+ respectively.

Table 4.7 reports the highest F_1 measure of each algorithm and their corresponding running times. TEA+ consistently produces clusters with the best average F_1 -measures and least running times for all datasets except *DBLP*. On *DBLP*, TEA produces clusters with the best average F_1 -measure while TEA+ produces clusters with slightly smaller F_1 measure but significantly faster. We also observe that ClusterHKPR and Monte-Carlo generate very similar results for all datasets. They run significantly slower than TEA and TEA+ and also produce clusters with slightly smaller average F_1 -measures than our methods. In addition, HK-Relax has the worst performance on most datasets. The only exception occurs on *Orkut*, where it produces clusters with the second best F_1 -measure but $4\times$ slower than TEA+.

4.4.7 Sensitivity Analysis to the Subgraph Characteristics

Next, we study the impact of query sets generated from subgraphs of different characteristics on clustering quality and efficiency. First, from each dataset of *Youtube*, *PLC* and *Orkut*, we select 250 subgraphs with different densities [89] randomly. Then we sort the subgraphs by their densities in descending order (denoted as $\{SG_1, SG_2, \dots, SG_{250}\}$). We pick 50 nodes from $SG_1, \dots, SG_{50}$ respectively to form a query set referred to as *high-density* seed nodes, 50 nodes from $SG_{100}, \dots, SG_{150}$ respectively as *medium-density* seed nodes, and 50 nodes from $SG_{200}, \dots, SG_{250}$ respectively as *low-density* seed nodes. We run ClusterHKPR, Monte-Carlo, HK-Relax, TEA and TEA+ with the same parameter settings as in Section 4.4.4 on these three query sets.

Figure 4.7 plots the average conductance of the output clusters and the average running times of all algorithms under different error thresholds for the three query sets. We report the results on *DBLP*, *Youtube*, *PLC* and *Orkut* here. We can make the following observations. First, TEA and TEA+ are consistently faster than the existing approaches for all query sets. Second, the conductance values of each graph in Figures 4.7(i)-(l) are higher than the rest. This is because subgraphs with high densities have low conductance. Also, both ClusterHKPR and Monte-Carlo show similar results on all query sets for all datasets whereas HK-Relax, TEA and TEA+ are sensitive to the subgraph densities. Since seed nodes picked from subgraphs with high densities would have many neighbors, the residues in HK-Relax, TEA and TEA+ will drop quickly as push operations are performed, making them terminate quickly.

4.4.8 Effects of Heat Constant t

Lastly, we investigate the impact of the heat constant t . Using the same parameter settings and query set in Section 4.4.4, we run all algorithms on *DBLP* and *PLC* datasets by varying t in $\{5, 10, 20, 40\}$. Figures 4.8 and 4.9 plot the average running time and average conductance of the output clusters of each algorithm on *DBLP* and *PLC* respectively. Observe that the running time of each algorithm increases as we increase t , which is consistent with their time complexities. ClusterHKPR and Monte-Carlo are the slowest as t changes. We further observe that the advantage of TEA+ over competing methods is more prominent as t becomes larger. More specifically, TEA+ is around 4 times faster than HK-Relax when $t = 5$ and the speedup goes up to more than one order of magnitude when $t = 40$ on *DBLP*. Similar observations can be made on *PLC*. In addition, we find that the conductance values of clusters produced by each algorithm with larger t are smaller than those with smaller t . This shows that we can obtain clusters with small conductance by choose a large t . However, our “Clusters Produced vs. Ground Truth” experiment (see Section 4.4.6) reveals that clusters produced by all algorithms with a large t are very different from the ground-truth. This is because algorithms with a large t tend to give us a cluster of nodes that are far from the seed node. As a result, choosing a good t is paramount for finding high quality clusters.

4.5 Summary

In this chapter, we propose two novel heat-kernel-based local clustering algorithms, **TEA** and **TEA+**, for computing approximate HKPR values and local graph clustering efficiently. Our algorithms bridge deterministic graph traversal with Monte-Carlo random walks in a non-trivial way, thereby overcoming the drawbacks of both and achieving significant gain in performance in comparison to the state-of-the-art local clustering techniques. Our experiments demonstrate that **TEA+** significantly outperforms the state-of-the-art heat-kernel-based algorithm by at least 4 times on small graphs and up to one order of magnitude on large graphs in terms of computational time when producing clusters with the same qualities.

Chapter 5

Homogeneous Network Embedding via Node-Reweighted PageRank¹

In this chapter, we investigate the *homogeneous network embedding* problem, which arises as a very popular topic with applications in many graph analysis tasks such as link prediction, graph reconstruction, node classification, and node clustering. Given an input graph G and a node $v \in G$, HNE aims to map the graph structure in the vicinity of v to a fixed-dimensional feature vector. Such feature vectors can then be fed to a machine learning pipeline as inputs, *e.g.*, for link prediction, graph reconstruction, and node classification. The vast majority of existing approaches to HNE are either immensely expensive, and, thus, are limited to small graphs. The few that do scale to large graphs, on the other hand, are based on quick-and-rough heuristics that or fail to fully capture the local graph structure, leading to limited effectiveness of the extracted feature vectors. Meanwhile, in recent years there have been rapid advancements in scalable algorithms for computing approximate *personalized PageRank*, which captures rich graph topological information. However, PPR was designed for a very different purpose, *i.e.*, ranking nodes in G based on their relative importance from a source node’s perspective. In contrast, HNE aims to build node embeddings considering the whole graph. Consequently, node embeddings derived directly from PPR are of suboptimal utility. A natural idea is: can we construct an effective embedding of a node v based on its approximate PPR values? This turns out to be challenging, due to the fact that PPR was designed for a very different purpose, *i.e.*, ranking nodes in the graph based on their relative importance

¹The work in this chapter has been published as [168] in PVLDB 2020.

from an individual node v 's perspective. In contrast, in HNE the goal is to build unified node embeddings for all nodes from a global view. Consequently, using PPR values directly as node embeddings leads to sub-optimal result effectiveness. Motivated by this, we propose Node-Reweighted PageRank (NRP), a novel solution that transforms PPR values into effective node embeddings, by iteratively solving an optimization problem. NRP computation is efficient and scalable: in particular, NRP embeddings for a billion-edge graph can be computed in a few hours on a single CPU core. At the same time, the quality of NRP embeddings is highly competitive: as our experiments demonstrate, in terms of prediction accuracy, NRP consistently outperforms existing HNE methods on three common tasks: link prediction, graph reconstruction and node classification, on both large and small graphs.

The rest of the chapter is organized as follows. Section 5.1 presents the problem statement as well as the objective of our PPR-based solution. In Section 5.2 and Section 5.3, we present our proposed Baseline and NRP algorithms, respectively. In Section 5.4, we conduct comprehensive experiments to evaluate the effectiveness and efficiency of NRP against 18 recent HNE solutions. Finally, we briefly summarize this chapter in Section 5.5. Table 5.1 summarizes frequent notations used throughout the chapter.

5.1 Problem Formulation

This section introduces the *homogeneous network embedding* problem, followed by an overview of the high-level idea and objective of our HNE solution.

5.1.1 Homogeneous Network Embedding

As mentioned in Chapter 1, given an input graph $G = (V, E)$, the goal of HNE is to construct a size- k embedding for each node $v \in G$, where k is a user-specified per-node space budget. The input graph G can be either directed or undirected. For simplicity, in the following we assume that G is directed; for an undirected graph, we simply replace each undirected edge (u, v) with two directed ones with opposing direction, *i.e.*, (u, v) and (v, u) . Note that the capability to handle directed graphs is an advantage of our methods, compared to existing solutions that are limited to undirected graphs, *e.g.*, [99, 114, 182, 183].

Table 5.1: Frequently used notations.

Notation	Description
k	The dimensionality of the embedding vectors
$\mathbf{X}, \mathbf{Y}$	The forward and backward embeddings, respectively
$\vec{w}_v, \overleftarrow{w}_v$	The forward and backward weights for v 's forward and backward embeddings, respectively

In a directed graph, each node plays two roles: as the incoming end and outgoing end of edges, respectively. These two roles can have very different semantics. For instance, in a social graph, a computer scientist might be highly interested in coffee, and, thus, follows many coffee-related nodes. On the other hand, the same person is probably better known for her work in computer science rather than her interest in coffee. This motivates building two separate embedding vectors $\mathbf{X}_v$ and $\mathbf{Y}_v$ for each node v , referred to as the *forward* and *backward* embeddings of v , respectively. In our solutions, we assign equal space budget (*i.e.*, $\frac{k}{2}$) to $\mathbf{X}_v$ and $\mathbf{Y}_v$.

5.1.2 Solution Objective

Following the methodology of constructing embeddings through factorizing a proximity matrix $\mathbf{M}$ (described in Section 3.2), a straightforward and promising idea is to use PPR as the proximity measure. Specifically, the definition of PPR is based on random walks, as follows. Suppose that we start a random walk from a source node u . At each step, we terminate the walk with probability α , and continues the walk (*i.e.*, moving on to a random out-neighbor of u) with probability $1 - \alpha$. Then, for each node $v \in G$, we define its PPR value $\pi(u, v)$ with respect to the source node u as the probability that the walk originating from u terminates at v .

Formally, let $\mathbf{\Pi}$ be an $n \times n$ matrix where $\mathbf{\Pi}[i, j] = \pi(v_i, v_j)$ for the i -th node v_i and j -th node v_j in G , and $\mathbf{P}$ be the probability transition matrix of G , *i.e.*, $\mathbf{P}[i, j] = \frac{1}{d_{out}(v_i)}$, where v_j is an out-neighbor of v_i and $d_{out}(v_i)$ denotes the out-degree of v_i . Then,

$$\mathbf{\Pi} = \sum_{i=0}^{\infty} \alpha(1 - \alpha)^i \cdot \mathbf{P}^i. \quad (5.1)$$

If we directly use $\mathbf{\Pi}$ as the proximity matrix, *i.e.*, $\mathbf{M} = \mathbf{\Pi}$. The goal is then to factorize $\mathbf{\Pi}$ into the forward and backward embeddings of nodes of the input graph G , such that for each pair of nodes u and v , we have:

$$\mathbf{X}_u \mathbf{Y}_v^\top \approx \pi(u, v) \quad (5.2)$$

Note that directly computing $\mathbf{\Pi}$ (and subsequently factorizing it into the node embeddings $\mathbf{X}$ and $\mathbf{Y}$) is infeasible for a large graph. In particular, $\mathbf{\Pi}$ is a dense matrix that requires $O(n^2)$ space for n nodes, and Equation (5.1) involves summing up an infinite series. To alleviate this problem, we could apply an approximate PPR algorithm (reviewed in Section 3.2) to compute the top- L largest PPR values for each node in G , which reduces the space overhead to $O(nL)$. Unfortunately, even the state-of-the-art approximate top- L PPR algorithm, *i.e.*, **TopPPR**, is insufficient for our purpose. Specifically, **TopPPR** takes $O\left(\frac{L^{\frac{1}{4}} n^{\frac{3}{4}} \log n}{\sqrt{gap_\rho}}\right)$ time to compute the approximate top- L PPR values for each node, where $gap_\rho \leq 1$ is a parameter that quantifies the difference between the top- L and non-top- L PPR values [155]. To approximate the entire $\mathbf{\Pi}$, we would need to invoke **TopPPR** for every node, which incurs time super-quadratic to n . Empirically, Ref. [155] reports that running **TopPPR** on a billion-edge Twitter graph (used in our experiments as well) takes about 15 seconds CPU time, for $L = 500$. The same graph contains over 41 million nodes, meaning that running **TopPPR** on each of them would cost over 19 years of CPU time, which is infeasible even for a powerful computing cluster. We address this challenge in the next section with a simple and effective solution.

5.2 The Baseline Algorithm

This section presents **Baseline**, an efficient and scalable approach to HNE, which constructs node embeddings using approximate PPR as the proximity measure. **Baseline** is the foundation of our main proposal **NRP**, presented in Section 5.3. In what follows, Section 5.1 overviews **Baseline** and formally defines the main concepts. In the following, we present our scalable approximate PPR factorization algorithm.

Observe that our goal is to obtain the node embeddings $\mathbf{X}$ and $\mathbf{Y}$, rather than the PPR matrix $\mathbf{\Pi}$ itself. Thus, the main idea of **Baseline** is to integrate the computation and factorization of $\mathbf{\Pi}$ in the same iterative algorithm. Specifically, according to Equation (5.1), $\mathbf{\Pi}$ can be viewed as the weighted sum of proximity values of different orders, *i.e.*, one-hop proximity, two-hop proximity, etc. Therefore, instead of first computing $\mathbf{\Pi}$ and

then factorizing it into node embeddings, we can instead start by factorizing the first-order proximity matrix (*i.e.*, $\mathbf{P}$) into the initial embeddings $\mathbf{X}$ and $\mathbf{Y}$, and then iteratively refine $\mathbf{X}$ and $\mathbf{Y}$, thereby incorporating higher-order information into them.

First, we consider a *truncated* version of $\mathbf{\Pi}$ as follows:

$$\mathbf{\Pi}' = \sum_{i=1}^{\ell_1} \alpha(1-\alpha)^i \cdot \mathbf{P}^i, \quad (5.3)$$

where ℓ_1 is a relative large constant (*e.g.*, $\ell_1 = 20$). In other words, we set

$$\mathbf{\Pi}' = \mathbf{\Pi} - \alpha \mathbf{I} - \left(\sum_{i=\ell_1+1}^{+\infty} \alpha(1-\alpha)^i \cdot \mathbf{P}^i \right),$$

where $\mathbf{I}$ denotes an $n \times n$ identity matrix. The rationale is that when i is sufficiently large, $\alpha(1-\alpha)^i$ is small, in which case $\sum_{i=\ell_1+1}^{+\infty} \alpha(1-\alpha)^i \cdot \mathbf{P}^i$ becomes negligible. In addition, $\alpha \mathbf{I}$ only affects the PPR $\pi(u, u)$ from each node u to itself, which has no impact on our objective in Equation (5.2) since we only concern the PPR values between different nodes.

To decompose $\mathbf{\Pi}'$, observe that

$$\mathbf{\Pi}' = \left(\sum_{i=1}^{\ell_1} \alpha(1-\alpha)^i \cdot \mathbf{P}^{i-1} \right) \mathbf{D}^{-1} \mathbf{A},$$

where $\mathbf{A}$ is the adjacency matrix of G , and $\mathbf{D}$ is an $n \times n$ diagonal matrix where the i -th diagonal element is $d_{out}(v_i)$. Instead of applying exact singular value decomposition (SVD) that is very time consuming, we factorize $\mathbf{A}$ using the RandSVD algorithm in [104] for randomized SVD, obtaining two $n \times k'$ matrices $\mathbf{U}, \mathbf{V}$ and a $k' \times k'$ diagonal matrix $\mathbf{\Sigma}$ given inputs $\mathbf{A}$ and k' , such that $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top \approx \mathbf{A}$. In short, RandSVD reduces $\mathbf{A}$ to a low-dimensional space by the Gaussian random projection and then performs SVD on the low-dimensional matrix. Given a relative error threshold ϵ , RandSVD guarantees a $(1 + \epsilon)$ error bound for spectral norm low-rank approximation.

Given the output $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}$ from RandSVD, we set

$$\mathbf{X}_1 = \mathbf{D}^{-1} \mathbf{U} \sqrt{\mathbf{\Sigma}} \quad \text{and} \quad \mathbf{Y} = \mathbf{V} \sqrt{\mathbf{\Sigma}}.$$

We have $\mathbf{X}_1 \mathbf{Y}^\top \approx \mathbf{D}^{-1} \mathbf{A} = \mathbf{P}$. After that, we compute $\mathbf{X}_i = (1 - \alpha) \mathbf{P} \mathbf{X}_{i-1} + \mathbf{X}_1$ for $i = 2, \dots, \ell_1$, and we set $\mathbf{X} = \alpha(1 - \alpha) \mathbf{X}_{\ell_1}$. This results in $\mathbf{X} = \sum_{i=1}^{\ell_1} \alpha(1 - \alpha)^i \mathbf{P}^{i-1} \mathbf{X}_1$ and

$\mathbf{XY}^\top = \sum_{i=1}^{\ell_1} \alpha(1-\alpha)^i \mathbf{P}^{i-1} \cdot \mathbf{X}_1 \mathbf{Y}^\top$. It can be verified that $\mathbf{XY}^\top \approx \mathbf{\Pi}'$. Particularly, the following theorem establishes the accuracy guarantees of **Baseline**.

Theorem 5.1. *Given $\mathbf{A}, \mathbf{D}^{-1}, \mathbf{P}, k', \alpha, \ell_1$ and relative error threshold ϵ as inputs to Algorithm 6, it returns $\mathbf{X}$ and $\mathbf{Y}$ that satisfy, for every pair of nodes $(u, v) \in V \times V$ with $u \neq v$,*

$$\begin{aligned} & |\mathbf{\Pi}[u, v] - (\mathbf{XY}^\top)[u, v]| \\ & \leq (1 + \epsilon) \sigma_{k'+1} (1 - \alpha) (1 - (1 - \alpha)^{\ell_1}) + (1 - \alpha)^{\ell_1+1}, \end{aligned}$$

and for every node $u \in V$,

$$\begin{aligned} & \sum_{v \in V} |\mathbf{\Pi}[u, v] - (\mathbf{XY}^\top)[u, v]| \\ & \leq \sqrt{n} (1 + \epsilon) \sigma_{k'+1} (1 - \alpha) (1 - (1 - \alpha)^{\ell_1}) + (1 - \alpha)^{\ell_1+1}, \end{aligned}$$

where $\sigma_{k'+1}$ is the $(k' + 1)$ -th largest singular value of $\mathbf{A}$.

Proof. We need the following theorem,

Theorem 5.2 (Eckart–Young Theorem [57]). *Suppose $\mathbf{A}_{k'}$ is the rank k' approximation to $\mathbf{A}$ produced by exact SVD, then*

$$\min_{\text{rank}(\hat{\mathbf{A}}) \leq k'} \|\mathbf{A} - \hat{\mathbf{A}}\|_2 = \|\mathbf{A} - \mathbf{A}_{k'}\|_2 = \sigma_{k'+1}, \quad (5.4)$$

where $\sigma_{k'+1}$ is the $(k' + 1)$ -th largest singular value of $\mathbf{A}$.

Recall that $\mathbf{X}_1 \mathbf{Y}^\top = \mathbf{D}^{-1} \mathbf{U} \mathbf{\Sigma} \mathbf{V}^\top$, where $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}$ are produced by RandSVD. Then, by Theorem 1 of RandSVD [104] and Eckart–Young theorem [57], we have

$$\|\mathbf{A} - \mathbf{U} \mathbf{\Sigma} \mathbf{V}^\top\|_2 = \|\mathbf{A} - \mathbf{D} \mathbf{X}_1 \mathbf{Y}^\top\|_2 \leq (1 + \epsilon) \sigma_{k'+1}, \quad (5.5)$$

where $\sigma_{k'+1}$ is the $k' + 1$ largest singular value of $\mathbf{A}$. According to [57], the following inequalities hold

$$\begin{aligned} \|\mathbf{A} - \mathbf{D} \mathbf{X}_1 \mathbf{Y}^\top\|_{\max} & \leq \|\mathbf{A} - \mathbf{D} \mathbf{X}_1 \mathbf{Y}^\top\|_2 \leq (1 + \epsilon) \sigma_{k'+1}, \\ \|\mathbf{A} - \mathbf{D} \mathbf{X}_1 \mathbf{Y}^\top\|_1 & \leq \sqrt{n} \|\mathbf{A} - \mathbf{D} \mathbf{X}_1 \mathbf{Y}^\top\|_2 \leq \sqrt{n} (1 + \epsilon) \sigma_{k'+1}, \end{aligned}$$

which indicates that, for any node pair $(u, v) \in V \times V$,

$$\begin{aligned} |\mathbf{P}[u, v] - (\mathbf{X}_1 \mathbf{Y}^\top)[u, v]| &= \left| \frac{\mathbf{A}[u, v]}{d(u)} - (\mathbf{X}_1 \mathbf{Y}^\top)[u, v] \right| \\ &\leq \frac{1}{d(u)} (1 + \epsilon) \sigma_{k'+1}, \end{aligned} \quad (5.6)$$

and for any node $u \in V$,

$$\begin{aligned} \sum_{u \in V} |\mathbf{P}[u, v] - (\mathbf{X}_1 \mathbf{Y}^\top)[u, v]| &= \sum_{u \in V} \left| \frac{\mathbf{A}[u, v]}{d(u)} - (\mathbf{X}_1 \mathbf{Y}^\top)[u, v] \right| \\ &\leq \sqrt{n} (1 + \epsilon) \sigma_{k'+1}. \end{aligned} \quad (5.7)$$

By Lines 2-5 in Algorithm 6,

$$\mathbf{X} \mathbf{Y}^\top = \alpha (1 - \alpha) \mathbf{X}_{\ell_1} \mathbf{Y}^\top = \sum_{i=1}^{\ell_1} \mathbf{P}^{i-1} \mathbf{X}_1 \mathbf{Y}^\top. \quad (5.8)$$

By the definition of $\mathbf{\Pi}'$ in Equation (5.3),

$$\begin{aligned} |\mathbf{\Pi}'[u, v] - (\mathbf{X} \mathbf{Y}^\top)[u, v]| &= \left| \sum_{i=1}^{\ell_1} \alpha (1 - \alpha)^i \sum_{w \in V} \mathbf{P}^{i-1}[u, w] \cdot (\mathbf{P}[w, v] - (\mathbf{X} \mathbf{Y}^\top)[w, v]) \right| \end{aligned} \quad (5.9)$$

With Equations (5.6), (5.7) and (5.9), for every node pair $(u, v) \in V \times V$ with $v \neq u$ and every node $u \in V$, the follow inequalities hold,

$$\begin{aligned} |\mathbf{\Pi}'[u, v] - (\mathbf{X} \mathbf{Y}^\top)[u, v]| &\leq \sigma_{k'+1} (1 + \epsilon) \sum_{i=1}^{\ell_1} \alpha (1 - \alpha)^i. \\ \sum_{v \in V} |\mathbf{\Pi}'[u, v] - (\mathbf{X} \mathbf{Y}^\top)[u, v]| &\leq \sqrt{n} \sigma_{k'+1} (1 + \epsilon) \sum_{i=1}^{\ell_1} \alpha (1 - \alpha)^i. \end{aligned} \quad (5.10)$$

In addition, according to Equations (5.1) and (5.3), for every node pair $(u, v) \in V \times V$ with $v \neq u$, we have

$$\begin{aligned} |\mathbf{\Pi}[u, v] - \mathbf{\Pi}'[u, v]| &\leq \sum_{v \in V} |\mathbf{\Pi}[u, v] - \mathbf{\Pi}'[u, v]| \\ &\leq 1 - \sum_{i=0}^{\ell_1} \alpha (1 - \alpha)^i. \end{aligned} \quad (5.11)$$

Algorithm 6: Baseline

Input: $\mathbf{A}, \mathbf{D}^{-1}, \mathbf{P}, \alpha, k', \ell_1, \epsilon$.
Output: $\mathbf{X}, \mathbf{Y}$.
 1 $[\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}] \leftarrow \text{RandSVD}(\mathbf{A}, k', \epsilon)$;
 2 $\mathbf{X}_1 \leftarrow \mathbf{D}^{-1}\mathbf{U}\sqrt{\mathbf{\Sigma}}, \quad \mathbf{Y} \leftarrow \mathbf{V}\sqrt{\mathbf{\Sigma}}$;
 3 **for** $i \leftarrow 2$ **to** ℓ_1 **do**
 4 $\mathbf{X}_i \leftarrow (1 - \alpha)\mathbf{P}\mathbf{X}_{i-1} + \mathbf{X}_1$;
 5 $\mathbf{X} \leftarrow \alpha(1 - \alpha)\mathbf{X}_{\ell_1}$;
 6 **return** $\mathbf{X}, \mathbf{Y}$;

Combining Equations (5.10) and (5.11) obtains the following results, for every node pair $(u, v) \in V \times V$ with $v \neq u$,

$$\begin{aligned}
 & |\mathbf{\Pi}[u, v] - \mathbf{X}\mathbf{Y}^\top(u, v)| \\
 & \leq |\mathbf{\Pi}[u, v] - \mathbf{\Pi}'[u, v]| + |\mathbf{\Pi}'[u, v] - (\mathbf{X}\mathbf{Y}^\top)[u, v]| \\
 & \leq (1 + \epsilon)\sigma_{k'+1}(1 - \alpha) (1 - (1 - \alpha)^{\ell_1}) + (1 - \alpha)^{\ell_1+1},
 \end{aligned}$$

and for every node $u \in V$,

$$\begin{aligned}
 & \sum_{v \in V} |\mathbf{\Pi}[u, v] - \mathbf{X}\mathbf{Y}^\top(u, v)| \\
 & \leq \sum_{v \in V} |\mathbf{\Pi}[u, v] - \mathbf{\Pi}'[u, v]| + \sum_{v \in V} |\mathbf{\Pi}'[u, v] - (\mathbf{X}\mathbf{Y}^\top)[u, v]| \\
 & \leq \sqrt{n}(1 + \epsilon)\sigma_{k'+1}(1 - \alpha) (1 - (1 - \alpha)^{\ell_1}) + (1 - \alpha)^{\ell_1+1},
 \end{aligned}$$

which completes our proof. $\square$

We then use $\mathbf{X}_v$ and $\mathbf{Y}_v$ as the initial forward and backward embeddings, respectively, for each node v . Algorithm 6 summarizes the pseudo-code for this construction of $\mathbf{X}$ and $\mathbf{Y}$. Next we present a concrete example.

Example 5.1. Given input graph G in Figure 1.1 and input parameters $k' = 2$, $\alpha = 0.15$, $\ell_1 = 20$, we run Algorithm 6 on G . It first applies RandSVD on the adjacency matrix $\mathbf{A} \in \mathbb{R}^{9 \times 9}$, which produces $\mathbf{X}_1 \in \mathbb{R}^{9 \times 2}$ and $\mathbf{Y} \in \mathbb{R}^{9 \times 2}$ as shown in Figure 5.1.

Baseline first sets $\mathbf{X} = \mathbf{X}_1$. Then, in each of the following iterations, the algorithm updates $\mathbf{X}$ to $0.85 \cdot \mathbf{P}\mathbf{X} + \mathbf{X}_1$. After repeating this process for $\ell_1 - 1 = 19$ iterations, Baseline scales $\mathbf{X}$ by the weight $\alpha(1 - \alpha) = 0.1275$ and returns us $\mathbf{X}$ and $\mathbf{Y}$ as in Figure 5.1:

$$\mathbf{Y} = \begin{bmatrix} \mathbf{Y}_{v_1} \\ \mathbf{Y}_{v_2} \\ \mathbf{Y}_{v_3} \\ \mathbf{Y}_{v_4} \\ \mathbf{Y}_{v_5} \\ \mathbf{Y}_{v_6} \\ \mathbf{Y}_{v_7} \\ \mathbf{Y}_{v_8} \\ \mathbf{Y}_{v_9} \end{bmatrix} = \begin{bmatrix} -0.081, & -0.933 \\ -1.211, & 0.104 \\ -0.081, & -0.933 \\ -1.211, & 0.104 \\ -0.110, & -1.277 \\ -0.846, & 0.073 \\ -0.053, & -0.617 \\ -0.594, & 0.051 \\ -0.026, & -0.296 \end{bmatrix}, \mathbf{X}_1 = \begin{bmatrix} -0.467, & 0.040 \\ -0.035, & -0.404 \\ -0.467, & 0.040 \\ -0.035, & -0.404 \\ -0.426, & 0.037 \\ -0.037, & -0.423 \\ -0.309, & 0.027 \\ -0.026, & -0.297 \\ -0.296, & 0.026 \end{bmatrix}, \dots, \mathbf{X} = \begin{bmatrix} \mathbf{X}_{v_1} \\ \mathbf{X}_{v_2} \\ \mathbf{X}_{v_3} \\ \mathbf{X}_{v_4} \\ \mathbf{X}_{v_5} \\ \mathbf{X}_{v_6} \\ \mathbf{X}_{v_7} \\ \mathbf{X}_{v_8} \\ \mathbf{X}_{v_9} \end{bmatrix} = \begin{bmatrix} -0.238, & -0.197 \\ -0.142, & -0.158 \\ -0.238, & -0.197 \\ -0.142, & -0.158 \\ -0.168, & -0.121 \\ -0.184, & -0.187 \\ -0.170, & -0.139 \\ -0.129, & -0.147 \\ -0.253, & -0.247 \end{bmatrix}$$

Figure 5.1: Illustration of Example 5.1 for the Baseline algorithm.

The inner product between $\mathbf{X}_u$ and $\mathbf{Y}_v$ approximates $\pi(u, v)$. For example, consider node pairs $\langle v_2, v_4 \rangle$ and $\langle v_9, v_7 \rangle$:

$$\begin{aligned} \mathbf{X}_{v_2} \mathbf{Y}_{v_4}^\top &= [-0.142, -0.158] \cdot [-1.211, 0.104]^\top = 0.1555, \\ \mathbf{X}_{v_9} \mathbf{Y}_{v_7}^\top &= [-0.253, -0.247] \cdot [-0.053, -0.617]^\top = 0.1658, \end{aligned}$$

which are close to $\pi(v_2, v_4)$ and $\pi(v_9, v_7)$ in Table 1.1 respectively. $\square$

Time Complexity. By the analysis in [104], applying RandSVD over sparse matrix $\mathbf{A}$ requires $O\left((mk' + nk'^2)\frac{\log n}{\epsilon}\right)$ time, where ϵ is a constant that controls the tradeoff between the efficiency and accuracy of SVD. In addition, Lines 2, 4, and 5 in Algorithm 6 respectively run in $O(mk')$ time. Therefore, the overall time complexity of Algorithm 6 is $O\left(\left(\frac{\log n}{\epsilon} + \ell_1\right)mk' + \frac{\log n}{\epsilon}nk'^2\right)$, which equals $O(k(m + kn)\log n)$ when ϵ and ℓ_1 are constants.

5.3 Proposed NRP Algorithm

The Baseline algorithm presented in the previous section directly uses PPR as the proximity measure. However, as explained in Chapter 1, PPR by itself is not suitable for our purpose since it is a local measure, in the sense that PPR values are *relative* with respect to the source node. Consequently, PPR values for different source nodes are essentially incomparable, which is the root cause of the counter-intuitive observation in the example of Figure 1.1 and Table 1.1.

In the proposed algorithm **NRP**, we address the deficiency of PPR through a technique that we call *node re-weighting*. Specifically, for any two nodes u and v , we aim to find forward and backward embeddings such that:

$$\mathbf{X}_u \mathbf{Y}_v^\top \approx \vec{w}_u \cdot \pi(u, v) \cdot \overleftarrow{w}_v \quad (5.12)$$

where $\pi(u, v)$ is the PPR value of v with respect to node u as source, $\vec{w}_u$ and $\overleftarrow{w}_v$ are weights assigned to u and v , respectively. In other words, we let $\mathbf{X}_u^\top \mathbf{Y}_v$ preserve a scaled version of $\pi(u, v)$. The goal of **NRP** is then to find approximate node weights so that Equation (5.12) properly expresses the proximity between nodes u and v . In **NRP**, the node weights are learned from data through an efficient optimization algorithm, described later in this section.

In the following, Section 5.3.1 explains the choice of node weights in **NRP**. Sections 5.3.2 and 5.3.3 elaborate on the computation of node weights. Section 5.3.5 summarizes the complete **NRP** algorithm.

5.3.1 Choice of Node Weights

As discussed before, the problem of PPR as a proximity measure is that it is a relative measure with respect to the source node. In particular, the PPR value does not take into account the number of out-going and in-coming edges that each node has. To address this issue, **NRP** assigns to each node u a *forward weight* $\vec{w}_u$ and a *backward weight* $\overleftarrow{w}_u$, and uses $\vec{w}_u \cdot \pi(u, v) \cdot \overleftarrow{w}_v$ instead of $\pi(u, v)$ to gauge the strength of connection from u to v , as in Equation (5.12). To compensate for the lack of node degree information in PPR values, we choose the forward and backward weights such that

$$\begin{aligned} \forall u \in V, \quad \sum_{\forall v \in V \setminus u} (\vec{w}_u \cdot \pi(u, v) \cdot \overleftarrow{w}_v) &\approx d_{out}(u), \text{ and} \\ \forall v \in V, \quad \sum_{\forall u \in V \setminus v} (\vec{w}_u \cdot \pi(u, v) \cdot \overleftarrow{w}_v) &\approx d_{in}(v). \end{aligned} \quad (5.13)$$

In other words, for any nodes $u, v \in G$, we aim to ensure that (i) the “total strength” of connections from u to other nodes is roughly equal to the out-degree $d_{out}(u)$ of u , and (ii) the total strength of connection from other nodes to v is roughly to equal the in-degree $d_{in}(v)$ of v . The rationale is that if u has a large out-degree, then it is more

likely to be connected to other nodes, and hence, the proximity from u to other nodes should be scaled up accordingly. The case for a node v with a large in-degree is similar. In Section 5.4, we empirically show that this scaling approach significantly improves the effectiveness of our embeddings for not just link prediction but also other important graph analysis task such as graph reconstruction.

5.3.2 Learning Node Weights

Given the output $\mathbf{X}$ and $\mathbf{Y}$ of Baseline (Algorithm 6), we use $\mathbf{X}_v \mathbf{Y}_v^\top$ as an approximation of $\pi(u, v)$ for any two different nodes u and v . Based on this, we formulate an objective function O for tuning node weights according to Equation (5.13):

$$\begin{aligned}
 O = \min_{\vec{w}, \overleftarrow{w}} \sum_v & \left\| \sum_{u \neq v} (\vec{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v) - d_{in}(v) \right\|_2 \\
 & + \sum_u \left\| \sum_{v \neq u} (\vec{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v) - d_{out}(u) \right\|_2 \\
 & + \lambda \sum_u (\|\vec{w}_u\|_2 + \|\overleftarrow{w}_u\|_2), \\
 & \text{subject to } \forall u \in V, \vec{w}_u, \overleftarrow{w}_u \geq \frac{1}{n}.
 \end{aligned} \tag{5.14}$$

To explain, recall that we use $\vec{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v$ to quantify the strength of connection from u to v , and hence, for any fixed u (resp. v), $\sum_{v \neq u} (\vec{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v)$ measures the total strength of connections from u to other nodes (resp. from other nodes to v). Therefore, by minimizing O , we aim to ensure that the total strength of connections starting from (resp. ending at) each node u is close to u 's out-degree (resp. in-degree), subject to a regularization term $\lambda \sum_u (\|\vec{w}_u\|_2 + \|\overleftarrow{w}_u\|_2)$. In addition, we require that $\vec{w}_u, \overleftarrow{w}_u \geq \frac{1}{n}$ for all nodes u to avoid negative node weights.

We derive an approximate solution for Equation (5.14) using coordinate descent [156]: We start with an initial solution $\vec{w}_v = d_{out}(v)$ and $\overleftarrow{w}_v = 1$ for each node v , and then iteratively update each weight based on the other $2n - 1$ weights. In particular, for any node v^* , the formula for updating $\overleftarrow{w}_{v^*}$ is derived by taking the partial derivative of the

objective function in Equation (5.14) with respect to $\overleftarrow{w}_{v^*}$:

$$\begin{aligned}
 \frac{\partial O}{\partial \overleftarrow{w}_{v^*}} &= 2 \left[\left(\left(\sum_{u \neq v^*} \vec{w}_u \mathbf{X}_u \right) \mathbf{Y}_{v^*}^\top \right)^2 \overleftarrow{w}_{v^*} \right. \\
 &\quad - d_{in}(v^*) \left(\sum_{u \neq v^*} \vec{w}_u \mathbf{X}_u \right) \mathbf{Y}_{v^*}^\top \\
 &\quad + \sum_u \left(\sum_{v \neq u, v \neq v^*} \vec{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v \right) \vec{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top \\
 &\quad + \sum_{u \neq v^*} (\vec{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top)^2 \overleftarrow{w}_{v^*} \\
 &\quad \left. - \left(\sum_u d_{out}(u) \vec{w}_u \mathbf{X}_u \right) \mathbf{Y}_{v^*}^\top + \lambda \overleftarrow{w}_{v^*} \right] \\
 &= 2(a_3 - a_2 - a_1) + 2(b_1 + b_2 + \lambda) \overleftarrow{w}_{v^*},
 \end{aligned}$$

where

$$\begin{aligned}
 a_1 &= \left(\sum_u d_{out}(u) \vec{w}_u \mathbf{X}_u \right) \mathbf{Y}_{v^*}^\top, \\
 a_2 &= d_{in}(v^*) \left(\sum_{u \neq v^*} \vec{w}_u \mathbf{X}_u \right) \mathbf{Y}_{v^*}^\top, \\
 a_3 &= \sum_u \left(\sum_{v \neq u, v \neq v^*} \vec{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v \right) \vec{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top, \\
 b_1 &= \sum_{u \neq v^*} (\vec{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top)^2, \\
 b_2 &= \left(\left(\sum_{u \neq v^*} \vec{w}_u \mathbf{X}_u \right) \mathbf{Y}_{v^*}^\top \right)^2.
 \end{aligned} \tag{5.15}$$

We identify the value of $\overleftarrow{w}_{v^*}$ that renders the above partial derivative zero, *i.e.*, $\frac{\partial O}{\partial \overleftarrow{w}_{v^*}} = 0$. If the identified $\overleftarrow{w}_{v^*}$ is smaller than $\frac{1}{n}$, then we set it to $\frac{1}{n}$ instead to avoid negativity. This leads to the following formula for updating backward weight $\overleftarrow{w}_{v^*}$:

$$\overleftarrow{w}_{v^*} = \max \left\{ \frac{1}{n}, \frac{a_1 + a_2 - a_3}{b_1 + b_2 + \lambda} \right\} \tag{5.16}$$

By Equation (5.16), each update of $\overleftarrow{w}_{v^*}$ requires computing a_1, a_2, a_3, b_1 and b_2 . Towards this end, a straightforward approach is to compute these variables directly based

on their definitions in Equation (5.15). This, however, leads to tremendous overheads. In particular, computing a_1 , a_2 , and b_2 requires a linear scan of $\mathbf{X}_u$ for each node u , which requires $O(nk')$ time. Deriving b_1 requires computing $\vec{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top$ for each node u , which incurs $O(nk'^2)$ overhead. Furthermore, computing b_3 requires calculating $\vec{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v$ for all $u \neq v \neq v^*$, which takes $O(n^2k'^2)$ time. Therefore, each update of $\overleftarrow{w}_{v^*}$ takes $O(n^2k'^2)$, which leads to a total overhead of $O(n^3k'^2)$ for updating all $\overleftarrow{w}_{v^*}$ once. Apparently, this overhead is prohibitive for large graphs. To address this deficiency, in Section 5.3.3, we present a solution that reduces the overhead to $O(nk'^2)$ instead of $O(n^3k'^2)$.

5.3.3 Accelerating Weight Updates

We observe that the updates of different node weights share a large amount of common computation. For example, for any node v^* , deriving a_1 always requires computing $\sum_u d_{out}(u) \vec{w}_u \mathbf{X}_u$. Intuitively, if we are able to reuse the result of such common computation for different nodes, then the overheads of our coordinate descent algorithm could be significantly reduced. In what follows, we elaborate how we exploit this idea to accelerate the derivation of a_1, a_2, a_3, b_1 , and b_2 .

Computation of $\mathbf{a}_1, \mathbf{a}_2, \mathbf{b}_2$. By the definitions of a_1, a_2, b_2 in Equation (5.15),

$$\begin{aligned} a_1 &= \boldsymbol{\xi} \mathbf{Y}_{v^*}^\top, \quad a_2 = d_{in}(v^*) (\boldsymbol{\chi} - \vec{w}_{v^*} \mathbf{X}_{v^*}) \mathbf{Y}_{v^*}^\top, \\ \text{and } b_2 &= ((\boldsymbol{\chi} - \vec{w}_{v^*} \mathbf{X}_{v^*}) \mathbf{Y}_{v^*}^\top)^2, \\ \text{where } \boldsymbol{\xi} &= \sum_u d_{out}(u) \vec{w}_u \mathbf{X}_u, \quad \text{and } \boldsymbol{\chi} = \sum_u \vec{w}_u \mathbf{X}_u. \end{aligned} \tag{5.17}$$

Equation (5.17) indicates that the a_1 values of all nodes $v^* \in V$ share a common $\boldsymbol{\xi}$, while a_2 and b_2 of each node v^* have $\boldsymbol{\chi}$ in common. Observe that both $\boldsymbol{\xi}$ and $\boldsymbol{\chi}$ are independent of any backward weight. Motivated by this, we propose to first compute $\boldsymbol{\xi} \in \mathbb{R}^{1 \times k'}$ and $\boldsymbol{\chi} \in \mathbb{R}^{1 \times k'}$, which takes $O(nk')$ time. After that, we can easily derive a_1, a_2 , and b_2 for any node with precomputed $\boldsymbol{\xi}$ and $\boldsymbol{\chi}$. In that case, each update of a_1, a_2 , and b_2 takes only $O(k')$ time, due to Equation (5.17). This leads to $O(nk')$ (instead of $O(n^2k')$) total computation time of a_1, a_2 , and b_2 for all nodes.

Computation of a_3 . Note that

$$\begin{aligned}
 a_3 = & \sum_u \left(\sum_v \vec{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v \right) \vec{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top \\
 & - \sum_u \left(\vec{w}_u \mathbf{X}_u \mathbf{Y}_{v^*}^\top \overleftarrow{w}_{v^*} \right) \vec{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \\
 & - \sum_v \left(\vec{w}_v \mathbf{X}_v \mathbf{Y}_v^\top \overleftarrow{w}_v \right) \vec{w}_v \mathbf{X}_v \mathbf{Y}_{v^*}^\top \\
 & + \left(\vec{w}_{v^*} \mathbf{X}_{v^*} \mathbf{Y}_{v^*}^\top \overleftarrow{w}_{v^*} \right) \vec{w}_{v^*} \mathbf{X}_{v^*} \mathbf{Y}_{v^*}^\top,
 \end{aligned}$$

which can be rewritten as:

$$\begin{aligned}
 a_3 = & \boldsymbol{\rho}_1 \boldsymbol{\Lambda} \mathbf{Y}_{v^*}^\top - \overleftarrow{w}_{v^*} \mathbf{Y}_{v^*} \boldsymbol{\Lambda} \mathbf{Y}_{v^*}^\top - \boldsymbol{\rho}_2 \mathbf{Y}_{v^*}^\top \\
 & + \overleftarrow{w}_{v^*} (\mathbf{X}_{v^*} \mathbf{Y}_{v^*}^\top)^2 \vec{w}_{v^*}^2, \\
 \text{where } \boldsymbol{\Lambda} = & \sum_u \vec{w}_u^2 (\mathbf{X}_u^\top \mathbf{X}_u), \quad \boldsymbol{\rho}_1 = \sum_v \overleftarrow{w}_v \mathbf{Y}_v, \\
 \text{and } \boldsymbol{\rho}_2 = & \sum_v (\vec{w}_v^2 \cdot \overleftarrow{w}_v (\mathbf{X}_v \mathbf{Y}_v^\top) \mathbf{X}_v).
 \end{aligned} \tag{5.18}$$

Observe that $\boldsymbol{\Lambda}$ is independent of any backward weight. Thus, it can be computed once and reused in the computation of a_3 for all nodes. Meanwhile, both $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ are dependent on all of the backward weights, and hence, cannot be directly reused if we are to update each backward weight in turn. However, we note that $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ can be *incrementally* updated after the change of any single backward weight. Specifically, suppose that we have computed $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ based on Equation (5.18), and then we change the backward weight of v^* from $\overleftarrow{w}'_{v^*}$ as $\overleftarrow{w}_{v^*}$. In that case, we can update $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ as:

$$\begin{aligned}
 \boldsymbol{\rho}_1 = & \boldsymbol{\rho}_1 + (\overleftarrow{w}_{v^*} - \overleftarrow{w}'_{v^*}) \mathbf{Y}_{v^*}, \\
 \boldsymbol{\rho}_2 = & \boldsymbol{\rho}_2 + (\overleftarrow{w}_{v^*} - \overleftarrow{w}'_{v^*}) \vec{w}_{v^*}^2 (\mathbf{X}_{v^*} \mathbf{Y}_{v^*}^\top) \mathbf{X}_{v^*}.
 \end{aligned} \tag{5.19}$$

Each of such updates takes only $O(k')$ time, since $\overleftarrow{w}_{v^*}, \overleftarrow{w}'_{v^*} \in \mathbb{R}$ and $\mathbf{X}_{v^*}, \mathbf{Y}_{v^*} \in \mathbb{R}^{1 \times k'}$.

The initial values of $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ can be computed in $O(nk')$ time based on Equation (5.18), while $\boldsymbol{\Lambda}$ can be calculated in $O(nk'^2)$ time. Given $\boldsymbol{\Lambda}$, $\boldsymbol{\rho}_1$, and $\boldsymbol{\rho}_2$, we can compute a_3 for any node v^* in $O(k'^2)$ time based on Equation (5.18). Therefore, the total time required for computing a_3 for all nodes is $O(nk'^2)$, which is a significant reduction from the $O(n^3k'^2)$ time required by the naive solution described in Section 5.3.2.

$$\begin{aligned}\boldsymbol{\xi} &= [-8.1453, -7.6509], \boldsymbol{\chi} = [-3.5227, -3.2933], \\ \boldsymbol{\rho}_1 &= [-4.2126, -3.7234], \boldsymbol{\rho}_2 = [-1.2659, -1.1678], \\ \boldsymbol{\Lambda} &= \begin{bmatrix} 1.4478, & 1.3308 \\ 1.3308, & 1.2575 \end{bmatrix}, \boldsymbol{\phi} = [1.4478, 1.2575].\end{aligned}$$

Figure 5.2: Illustration for Example 5.2

Approximation of b_1 . We observe that the value of b_1 is insignificant compared to b_2 . Thus, we propose to approximate its value instead of deriving it exactly, so as to reduce computation cost. By the inequality of arithmetic and geometric means, we have:

$$\frac{1}{k'} b_1 \leq \sum_{u \neq v^*} \vec{w}_u^2 \left(\sum_{r=1}^{k'} \mathbf{X}_u[r]^2 \mathbf{Y}_{v^*}[r]^2 \right) \leq b_1. \quad (5.20)$$

Let $\boldsymbol{\phi}$ be a length- k' vector where the r -th ($r \in [1, k']$) element is

$$\boldsymbol{\phi}[r] = \sum_u \vec{w}_u^2 \mathbf{X}_u[r]^2. \quad (5.21)$$

We compute $\boldsymbol{\phi}$ in $O(nk')$ time, and then, based on Equation (5.20), we approximate b_1 for each node in $O(k')$ time with

$$b_1 \approx \frac{k'}{2} \sum_{r=1}^{k'} \mathbf{Y}_{v^*}[r]^2 (\boldsymbol{\phi}[r] - \vec{w}_{v^*}^2 \mathbf{X}_{v^*}[r]^2). \quad (5.22)$$

Therefore, the total cost for approximating b_1 for all nodes is $O(nk')$.

Summary. As a summary, Algorithm 7 presents the pseudo-code of our method for updating the backward weight of each node. The algorithm first computes $\boldsymbol{\xi}, \boldsymbol{\chi}, \boldsymbol{\rho}_1, \boldsymbol{\rho}_2, \boldsymbol{\Lambda}, \boldsymbol{\phi}$ in $O(nk'^2)$ time (Lines 1-3). After that, it examines each node's backward weight in random order, and computes a_1, a_2, a_3, b_1, b_2 by Equations (5.17), (5.18), and (5.22), which takes $O(k'^2)$ time per node (Line 5). Given a_1, a_2, a_3, b_1, b_2 , the algorithm updates the backward weight examined, and then updates $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ in $O(k')$ time (Lines 7-9). The total time complexity of Algorithm 7 is $O(nk'^2)$, which is significantly better than the $O(n^3k'^2)$ -time method in Section 5.3.2. We illustrate Algorithm 7 with an example.

Algorithm 7: updateBwdWeights

Input: $G, k', \vec{w}, \overleftarrow{w}, \mathbf{X}, \mathbf{Y}$.
Output: $\overleftarrow{w}$

- 1 Compute $\xi, \chi, \rho_1, \rho_2, \Lambda$, and Φ based on Equation (5.17), (5.18), and (5.21);
- 2 **for** $r \leftarrow 1$ **to** k' **do**
- 3 $\phi[r] = \sum_u \vec{w}_u^2 \mathbf{X}_u[r]^2$;
- 4 **for** $v^* \in V$ **in random order** **do**
- 5 Compute a_1, a_2, a_3, b_1, b_2 by Equation (5.17), (5.18), and (5.22);
- 6 $\overleftarrow{w}'_{v^*} = \overleftarrow{w}_{v^*}$;
- 7 $\overleftarrow{w}_{v^*} = \max \left\{ \frac{1}{n}, \frac{a_1 + a_2 - a_3}{b_1 + b_2 + \lambda} \right\}$;
- 8 $\rho_1 = \rho_1 + (\overleftarrow{w}_{v^*} - \overleftarrow{w}'_{v^*}) \mathbf{Y}_{v^*}$;
- 9 $\rho_2 = \rho_2 + (\overleftarrow{w}_{v^*} - \overleftarrow{w}'_{v^*}) \vec{w}_{v^*}^2 (\mathbf{X}_{v^*} \mathbf{Y}_{v^*}^\top) \mathbf{X}_{v^*}$
- 10 **return** $\overleftarrow{w}$;

Example 5.2. Suppose that we invoke Algorithm 7 given graph G in Figure 1.1, $k' = 2$, $\mathbf{X}$ and $\mathbf{Y}$ from Example 5.1, and the following $\overleftarrow{w}$ and $\vec{w}$:

$$\overleftarrow{w} = [1, 1, 1, 1, 1, 1, 1, 1, 1], \quad \vec{w} = [2, 3, 2, 3, 3, 2, 2, 2, 1].$$

The algorithm first computes $\xi, \chi, \rho_1, \rho_2, \Lambda$ and ϕ according to Equations (5.17), (5.18), and (5.21). Figure 5.2 shows the results.

Then, we update each backward weight in a random order with the above precomputed values. Let's pick $\overleftarrow{w}_{v_1}$ for the first update. According to Equations (5.17), (5.18) and (5.22), we do not need to perform summations over all 9 nodes as in Equation (5.15) but some multiplications between a 2×2 matrix and a length-2 vector, as well as inner products between length-2 vectors, yielding the following results fast:

$$\begin{aligned}
 a_1 &= \xi \mathbf{Y}_{v_1}^\top = 7.7968, \\
 a_2 &= 2(\chi - 2\mathbf{X}_{v_1}) \mathbf{Y}_{v_1}^\top = 5.903, \\
 a_3 &= \rho_1 \Lambda \mathbf{Y}_{v_1}^\top - \mathbf{Y}_{v_1} \Lambda \mathbf{Y}_{v_1}^\top - \rho_2 \mathbf{Y}_{v_1}^\top = 8.1324, \\
 b_1 &= \sum_{r=1}^2 \mathbf{Y}_{v_1}[r]^2 (\phi[r] - \vec{w}_{v_1}^2 \mathbf{X}_{v_1}[r]^2) = 0.9683, \\
 b_2 &= ((\chi - 2\mathbf{X}_{v_1}) \mathbf{Y}_{v_1}^\top)^2 = 8.7113.
 \end{aligned}$$

Let $\lambda = 0$. The backward weight for v_1 is updated as

$$\overleftarrow{w}_{v_1} = \max \left\{ \frac{1}{9}, \frac{a_1 + a_2 - a_3}{b_1 + b_2} \right\} = 0.5752,$$

and then ρ_1 and ρ_2 are updated accordingly with the updated $\overleftarrow{w}_{v_1}$ based on Equation (5.19) before proceeding to the next backward weight. $\square$

5.3.4 Updating Forward Weights

The forward weights $\overrightarrow{w}_{u^*}$ can be learned using an algorithm very similar to Algorithm 7. For brevity, we include the details in this section. For any node u^* , the formula for updating $\overrightarrow{w}_{u^*}$ is derived by (i) taking the partial derivative of the objective function in Equation (5.14) with respect to $\overrightarrow{w}_{u^*}$,

$$\begin{aligned} \frac{\partial O}{\partial \overrightarrow{w}_{u^*}} &= 2 \left[\left(\mathbf{X}_{u^*} \sum_{v \neq u^*} \overleftarrow{w}_v \mathbf{Y}_v^\top \right)^2 \overrightarrow{w}_{u^*} \right. \\ &\quad - d_{out}(u^*) \mathbf{X}_{u^*} \sum_{v \neq u^*} \overleftarrow{w}_v \mathbf{Y}_v^\top \\ &\quad + \sum_v \left(\sum_{u \neq v, u \neq u^*} \overrightarrow{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v \right) \mathbf{X}_{u^*} \mathbf{Y}_v^\top \overleftarrow{w}_v \\ &\quad + \sum_{v \neq u^*} (\mathbf{X}_{u^*} \mathbf{Y}_v^\top \overleftarrow{w}_v)^2 \overrightarrow{w}_{u^*} \\ &\quad \left. - \mathbf{X}_{u^*} \sum_v d_{in}(v) \overleftarrow{w}_v \mathbf{Y}_v^\top + \lambda \overrightarrow{w}_{u^*} \right] \\ &= 2(a'_3 - a'_2 - a'_1) + 2(b'_1 + b'_2 + \lambda) \overrightarrow{w}_{u^*}, \end{aligned}$$

and then (ii) identifying the value of $\overrightarrow{w}_{u^*}$ that renders the partial derivative equal to zero. In addition, if the identified $\overrightarrow{w}_{u^*}$ is smaller than $\frac{1}{n}$, then we set it to $\frac{1}{n}$ instead.

Then the forward weight learning rule is as in Equation (5.23):

$$\begin{aligned}
 \vec{w}_{u^*} &= \max \left\{ \frac{1}{n}, \frac{a'_1 + a'_2 - a'_3}{b'_1 + b'_2 + \lambda} \right\}, \text{ where} \\
 a'_1 &= \mathbf{X}_{u^*} \sum_v d_{in}(v) \overleftarrow{w}_v \mathbf{Y}_v^\top, \\
 a'_2 &= d_{out}(u^*) \mathbf{X}_{u^*} \sum_{v \neq u^*} \overleftarrow{w}_v \mathbf{Y}_v^\top, \\
 a'_3 &= \sum_v \left(\sum_{u \neq v, u \neq u^*} \vec{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v \right) \mathbf{X}_{u^*} \mathbf{Y}_v^\top \overleftarrow{w}_v, \\
 b'_1 &= \sum_{v \neq u^*} (\mathbf{X}_{u^*} \mathbf{Y}_v^\top \overleftarrow{w}_v)^2, \\
 b'_2 &= (\mathbf{X}_{u^*} \sum_{v \neq u^*} \overleftarrow{w}_v \mathbf{Y}_v^\top)^2.
 \end{aligned} \tag{5.23}$$

By Equation (5.23), each update of $\vec{w}_{u^*}$ requires computing a'_1, a'_2, a'_3, b'_1 and b'_2 , which are similar to the computation of a_1, a_2, a_3, b_1 and b_2 in Section 5.3.2. Hence, it takes $O(n^2 k'^2)$ time to update $\vec{w}_{u^*}$ once, which leads to a total overhead of $O(n^3 k'^2)$ for updating all $\vec{w}_{u^*}$ once.

In the following, we present the solution to accelerate the computation of $a'_1, a'_2, a'_3, b'_1, b'_2$ for forward weight $\vec{w}_{u^*}$. Since the techniques for updating forward weight $\vec{w}_{u^*}$ are similar to that for backward weights, for brevity, we use the same symbols to represent the intermediate computations of forward weights as those of backward weights.

Computation of a'_1, a'_2, b'_2 . By the definitions of a_1, a_2, b_2 in Equation (5.23),

$$\begin{aligned}
 a'_1 &= \mathbf{X}_{u^*} \boldsymbol{\xi}^\top, \quad a'_2 = d_{out}(u^*) \mathbf{X}_{u^*} (\boldsymbol{\chi} - \overleftarrow{w}_{u^*} \mathbf{Y}_{u^*})^\top, \\
 \text{and } b'_2 &= (\mathbf{X}_{u^*} (\boldsymbol{\chi} - \overleftarrow{w}_{u^*} \mathbf{Y}_{u^*})^\top)^2, \\
 \text{where } \boldsymbol{\xi} &= \sum_v d_{in}(v) \overleftarrow{w}_v \mathbf{Y}_v, \quad \text{and } \boldsymbol{\chi} = \sum_v \overleftarrow{w}_v \mathbf{Y}_v.
 \end{aligned} \tag{5.24}$$

Equation (5.24) indicates that the a'_1 values of all nodes $u^* \in V$ share a common $\boldsymbol{\xi}$, while a'_2 and b'_2 of each node u^* have $\boldsymbol{\chi}$ in common. Observe that both $\boldsymbol{\xi}$ and $\boldsymbol{\chi}$ are independent of any forward weight. Motivated by this, we propose to first compute $\boldsymbol{\xi} \in \mathbb{R}^{1 \times k'}$ and $\boldsymbol{\chi} \in \mathbb{R}^{1 \times k'}$, which takes $O(nk')$ time. After that, we can easily derive a'_1, a'_2 , and b'_2 for any node with precomputed $\boldsymbol{\xi}$ and $\boldsymbol{\chi}$. In that case, each update of a'_1, a'_2 , and b'_2 takes

only $O(k')$ time, due to Equation (5.24). This leads to $O(nk')$ (instead of $O(n^2k')$) total computation time of a'_1, a'_2 , and b'_2 for all nodes.

Computation of a'_3 . Note that

$$\begin{aligned} a'_3 = & \sum_v \left(\sum_u \vec{w}_u \mathbf{X}_u \mathbf{Y}_v^\top \overleftarrow{w}_v \right) \overleftarrow{w}_v \mathbf{X}_{u^*} \mathbf{Y}_v^\top \\ & - \sum_v (\vec{w}_{u^*} \mathbf{X}_{u^*} \mathbf{Y}_v^\top \overleftarrow{w}_v) \overleftarrow{w}_v \mathbf{X}_{u^*} \mathbf{Y}_v^\top \\ & - \sum_v (\vec{w}_v \mathbf{X}_v \mathbf{Y}_v^\top \overleftarrow{w}_v) \overleftarrow{w}_v \mathbf{X}_{u^*} \mathbf{Y}_v^\top \\ & + (\vec{w}_{u^*} \mathbf{X}_{u^*} \mathbf{Y}_{u^*}^\top \overleftarrow{w}_{u^*}) \overleftarrow{w}_{u^*} \mathbf{X}_{u^*} \mathbf{Y}_{u^*}^\top, \end{aligned}$$

which can be rewritten as:

$$\begin{aligned} a'_3 = & \boldsymbol{\rho}_1 \boldsymbol{\Lambda} \mathbf{X}_{u^*}^\top - \vec{w}_{u^*} \mathbf{X}_{u^*} \boldsymbol{\Lambda} \mathbf{X}_{u^*}^\top - \boldsymbol{\rho}_2 \mathbf{X}_{u^*}^\top \\ & + \overleftarrow{w}_{u^*}^2 (\mathbf{X}_{u^*} \mathbf{Y}_{u^*}^\top)^2 \vec{w}_{u^*}, \end{aligned} \quad (5.25)$$

where $\boldsymbol{\Lambda} = \sum_v \overleftarrow{w}_v^2 (\mathbf{Y}_v^\top \mathbf{Y}_v)$, $\boldsymbol{\rho}_1 = \sum_u \vec{w}_u \mathbf{X}_u$,
and $\boldsymbol{\rho}_2 = \sum_v (\vec{w}_v \cdot \overleftarrow{w}_v^2 (\mathbf{X}_v \mathbf{Y}_v^\top) \mathbf{Y}_v)$.

Observe that $\boldsymbol{\Lambda}$ is independent of any forward weight. Thus, it can be computed once and reused in the computation of a'_3 for all nodes. Meanwhile, both $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ dependent on all of the forward weights, and hence, cannot be directly reused if we are to update each forward weight in turn. However, we note that $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ can be *incrementally* updated after the change of any single forward weight. Specifically, suppose that we have computed $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ based on Equation (5.25), and then we change the forward weight of u^* from $\vec{w}'_{u^*}$ as $\vec{w}_{u^*}$. In that case, we can update $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ as:

$$\begin{aligned} \boldsymbol{\rho}_1 = & \boldsymbol{\rho}_1 + (\vec{w}_{u^*} - \vec{w}'_{u^*}) \mathbf{X}_{u^*}, \\ \boldsymbol{\rho}_2 = & \boldsymbol{\rho}_2 + (\vec{w}_{u^*} - \vec{w}'_{u^*}) \overleftarrow{w}_{u^*}^2 (\mathbf{X}_{u^*} \mathbf{Y}_{u^*}^\top) \mathbf{Y}_{u^*}. \end{aligned} \quad (5.26)$$

Each of such updates takes only $O(k')$ time, since $\vec{w}_{u^*}, \vec{w}'_{u^*} \in \mathbb{R}$ and $\mathbf{X}_{u^*}, \mathbf{Y}_{u^*} \in \mathbb{R}^{1 \times k'}$.

The initial values of $\boldsymbol{\rho}_1$ and $\boldsymbol{\rho}_2$ can be computed in $O(nk')$ time based on Equation (5.25), while $\boldsymbol{\Lambda}$ can be calculated in $O(nk'^2)$ time. Given $\boldsymbol{\Lambda}$, $\boldsymbol{\rho}_1$, and $\boldsymbol{\rho}_2$, we can compute a'_3 for any node u^* in $O(k'^2)$ time based on Equation (5.25). Therefore, the total

Algorithm 8: updateFwdWeights

Input: $G, k', \vec{w}, \overleftarrow{w}, \mathbf{X}, \mathbf{Y}$.
Output: $\vec{w}$

- 1 Compute $\xi, \chi, \rho_1, \rho_2, \Lambda$ based on Equation (5.24), (5.25);
- 2 **for** $r \leftarrow 1$ to k' **do**
- 3 $\phi[r] = \sum_v \overleftarrow{w}_v^2 \mathbf{Y}_v[r]^2$;
- 4 **for** $u^* \in V$ in random order **do**
- 5 Compute $a'_1, a'_2, a'_3, b'_1, b'_2$ by Equation (5.24), (5.25), and (5.29);
- 6 $\vec{w}'_{u^*} = \vec{w}_{u^*}$;
- 7 $\vec{w}_{u^*} = \max \left\{ \frac{1}{n}, \frac{a'_1 + a'_2 - a'_3}{b'_1 + b'_2 + \lambda} \right\}$;
- 8 $\rho_1 = \rho_1 + (\vec{w}_{u^*} - \vec{w}'_{u^*}) \mathbf{X}_{u^*}$;
- 9 $\rho_2 = \rho_2 + (\vec{w}_{u^*} - \vec{w}'_{u^*}) \overleftarrow{w}_{u^*}^2 (\mathbf{X}_{u^*} \mathbf{Y}_{u^*}^\top) \mathbf{Y}_{u^*}$
- 10 **return** $\vec{w}$;

time required for computing a'_3 for all nodes is $O(nk'^2)$, which is an significant reduction from the $O(n^3k'^2)$ time required by the naive solution in Equation (5.23).

Approximation of b'_1 . We observe that the value of b'_1 is insignificant compared to b'_2 . Thus, we propose to approximate its value instead of deriving it exactly, so as to reduce computation cost. By the inequality of arithmetic and geometric means, we have:

$$\frac{1}{k'} b'_1 \leq \sum_{v \neq u^*} \overleftarrow{w}_v^2 \left(\sum_{r=1}^{k'} \mathbf{X}_{u^*}[r]^2 \mathbf{Y}_v[r]^2 \right) \leq b'_1. \quad (5.27)$$

Let ϕ be a length- k' vector where the r -th ($r \in [1, k']$) element is

$$\phi[r] = \sum_v \overleftarrow{w}_v^2 \mathbf{Y}_v[r]^2. \quad (5.28)$$

We compute ϕ in $O(nk')$ time, and then, based on Equation (5.27), we approximate b'_1 for each node in $O(k')$ time with

$$b'_1 \approx \frac{k'}{2} \sum_{r=1}^{k'} \mathbf{X}_{u^*}[r]^2 (\phi[r] - \overleftarrow{w}_{u^*}^2 \mathbf{Y}_{u^*}[r]^2). \quad (5.29)$$

Therefore, the total cost for approximating b'_1 for all nodes is $O(nk')$.

Algorithm 8 illustrates the pseudo-code for updating forward weights, which is analogous to Algorithm 7. Based on the above analysis, it is easy to verify that it has the same time complexity and space overhead as Algorithm 7.

Algorithm 9: NRP

Input: Graph G , embedding dimensionality k , thresholds ℓ_1, ℓ_2 , random walk decay factor α and error threshold ϵ
Output: Embedding matrices $\mathbf{X}$ and $\mathbf{Y}$.

```

1  $k' \leftarrow k/2$ ;
2  $[\mathbf{X}, \mathbf{Y}] \leftarrow \text{Baseline}(\mathbf{A}, \mathbf{D}^{-1}, \mathbf{P}, \alpha, k', \ell_1, \epsilon)$ ;
3 for  $v \in V$  do
4    $\vec{w}_v = d_{out}(v)$ ,  $\overleftarrow{w}_v = 1$ ;
5 for  $l \leftarrow 1$  to  $\ell_2$  do
6    $\overleftarrow{\mathbf{w}} = \text{updateBwdWeights}(G, k', \vec{\mathbf{w}}, \overleftarrow{\mathbf{w}}, \mathbf{X}, \mathbf{Y})$ ;
7    $\vec{\mathbf{w}} = \text{updateFwdWeights}(G, k', \vec{\mathbf{w}}, \overleftarrow{\mathbf{w}}, \mathbf{X}, \mathbf{Y})$ ;
8 for  $v \in V$  do
9    $\mathbf{X}_v = \vec{w}_v \cdot \mathbf{X}_v$ ,  $\mathbf{Y}_v = \overleftarrow{w}_v \cdot \mathbf{Y}_v$ ;
10 return  $\mathbf{X}, \mathbf{Y}$ ;
    
```

5.3.5 Complete NRP Algorithm and Analysis

Algorithm 9 presents the pseudo-code for constructing embeddings with NRP. Given a graph G , embedding dimensionality k , random walk decay factor α , thresholds ℓ_1, ℓ_2 and relative error threshold ϵ , it first generates the initial embedding matrices $\mathbf{X}$ and $\mathbf{Y}$ using Algorithm 6 (Lines 1-2, see Section 5.2 for details). After that, it initializes the forward and backward weights for each node (Lines 3-4) and then applies coordinate descent to refine the weights (Lines 5-7). In particular, in each epoch of the coordinate descent, it first invokes Algorithm 7 to update each backward weight once (Line 6), and then applies a similar algorithm to update each forward weight once (Line 7, see Algorithm updateFwdWeights). The total number of epochs is controlled by ℓ_2 , which we set to $O(\log n)$ for efficiency. After the coordinate descent terminates, NRP multiplies the forward (resp. backward) embedding of each node by its forward (resp. backward) weight to obtain the final embeddings (Lines 8-9).

Complexity Analysis. NRP has three main steps: Algorithm 6, Algorithm 7, and Algorithm updateFwdWeights. By the analysis in Section 5.2, Algorithm 6 runs in $O(k(m + kn) \log n)$ time, and its space overhead is determined by the number of non-zero entries in the matrices, which is $O(m + nk)$. For Algorithm 7 and Algorithm updateFwdWeights, each epoch takes $O(nk'^2)$ time, as analyzed in Section 5.3.3. Hence,

the time complexities of Algorithm 7 and Algorithm `updateFwdWeights` are both $O(nk'^2)$ when the number of epochs ℓ_2 is a constant. In addition, the space costs of Algorithm 7 and Algorithm `updateFwdWeights` depend on the size of $\xi, \chi, \rho_1, \rho_2, \Lambda, \phi$ and the number of weights, which is bounded by $O(nk')$. As a result, the time complexity of Algorithm 9 is $O(k(m + kn) \log n)$ and its space complexity is $O(m + nk)$.

Comparison with concurrent work [172]. STRAP, which is concurrent to this work, uses a different algorithm for approximate PPR computation and (subsequently) factorization, and it does not apply re-weighting to adjust the embedding of each node. The space and time complexities of STRAP are $O(\frac{n}{\delta})$ and $O(\frac{m}{\delta} + nk^2)$, respectively, where δ is the error threshold for PPR values. In the literature of approximate PPR processing (e.g., in [150, 155]), δ is commonly set to $\frac{1}{n}$, which would lead to prohibitively high space (i.e., $O(n^2)$) and time ($O(mn + nk^2)$) complexities, respectively. Instead, in [172], STRAP fixes δ to a constant 10^{-5} , in order to handle large graphs. Even so, STRAP is still far more expensive than the proposed solution NRP, as we demonstrate in the experiments in the next section.

5.4 Experiments

We experimentally evaluate our proposed method, i.e., NRP, against 18 existing methods, including 4 classic ones and 14 recent ones, on three graph analysis tasks: link prediction, graph reconstruction, and node classification. We also study the efficiency of all methods and analyze the parameter choices of NRP. All experiments are conducted using a single thread on a Linux machine powered by an Intel Xeon(R) E5-2650 v2@2.60GHz CPU and 96GB RAM.

5.4.1 Experimental Settings

Datasets. We experiment with seven real networks that are used in previous work [61, 108, 134], including two billion-edge networks: directed network *Twitter* [37] and undirected network *Friendster* [166]. Table 5.2 shows the dataset statistics. For *Wiki*, *BlogCatalog*, *Youtube*, and *Orkut*, we use the node labels suggested in previous work

Table 5.2: Dataset statistics ($K = 10^3$, $M = 10^6$, $B = 10^9$).

Name	$ V $	$ E $	Type	#labels
<i>Wiki</i>	4.78K	184.81K	directed	40
<i>BlogCatalog</i>	10.31K	333.98K	undirected	39
<i>Youtube</i>	1.13M	2.99M	undirected	47
<i>TWeibo</i>	2.32M	50.65M	directed	100
<i>Orkut</i>	3.1M	234M	undirected	100
<i>Twitter</i>	41.6M	1.2B	directed	-
<i>Friendster</i>	65.6M	1.8B	undirected	-

[61, 110, 134]. For *TWeibo*, we collect its node tags from [83] and only keep the top 100 tags in the network, following the practice in [134].

Competitors. We evaluate NRP against eighteen existing methods, including four classic methods (*i.e.*, DeepWalk, node2vec, LINE and DNGR) and fourteen recent methods, many of which have not been compared against each other in previous work. To our knowledge, we are the first to systematically evaluate such a large number of existing network embedding techniques. We categorize the eighteen existing methods into four groups as follows:

1. factorization-based methods: AROPE [183], RandNE [182], NetSMF [114], ProNE [180], and STRAP [172];
2. random-walk-based methods: DeepWalk [110], LINE [130], node2vec [61], PBG [91], APP [185], and VERSE [134];
3. neural-network-based methods: DNGR [14], DRNE [136], GraphGAN [143], and GA [1];
4. other methods: RaRE [63], NetHiex [99] and GraphWave [34].

Parameter Settings. For NRP, we set $\ell_1 = 20$, $\ell_2 = 10$, $\alpha = 0.15$, $\epsilon = 0.2$, and $\lambda = 10$. Note that $\ell_1 = 20$ means up to 20-order proximities can be preserved in the embeddings, and most forward and backward weights converge with $\ell_2 = 10$ epochs. For fair comparison, the random walk decay factor, α , is set to 0.15, in all PPR-based methods, including VERSE, APP, STRAP, and NRP. We use the default parameter settings of all competitors as suggested in their papers. For instance, the error threshold δ in STRAP

is set to 10^{-5} as suggested in [172]. We obtain the source codes of all competitors from their respective authors. Unless otherwise specified, we set the embedding dimensionality k of each method to 128.

Note that AROPE, RandNE, NetHiex, GraphWave, NetSMF and ProNE are designed for undirected graphs only. For a thorough evaluation, we still report their performance on the directed graphs, *i.e.*, *Wiki*, *TWeibo*, and *Twitter*, by omitting the direction of each edge when feeding the graphs as input to these methods.

In addition, the following methods are designed for some specific tasks: *e.g.*, GraphWave and DRNE for structural role discovery; node2vec, DeepWalk, LINE, DNNGR, NetSMF and ProNE for node classification or network visualization. For completeness, we evaluate all methods over three commonly used tasks, namely, link prediction, node classification, and graph reconstruction. We exclude a method if it cannot report results within 7 days.

5.4.2 Link Prediction

Link prediction aims to predict which node pairs are likely to form edges. Following previous work [183], we first remove 30% randomly selected edges from the input graph G , and then construct embeddings on the modified graph G' . After that, we form a testing set E_{test} consisting of (i) the node pairs corresponding to the 30% removed edges, and (ii) an equal number of node pairs that are not connected by any edge in G . Note that on directed graphs, each node pair (u, v) is ordered, *i.e.*, we aim to predict whether there is a directed edge from u to v .

Given a method's embeddings, we compute a score for each node pair (u, v) in the testing set based on embedding vectors of u and v , and then evaluate the method's performance by the *Area Under Curve (AUC)* of the computed scores. Following their own settings, for AROPE, RandNE, NetHiex, NetSMF and ProNE, the score for (u, v) is computed as the inner product u and v 's embedding vectors; for NRP, Baseline, APP, GA, and STRAP, the score equals the inner product of u 's forward vector and v 's backward vector. For RaRE, we apply the probability function described in [62] for computing the score for (u, v) . For DeepWalk, LINE, node2vec, DNNGR, DRNE, GraphGAN, and GraphWave, we use the *edge features* approach [99]: (i) for each node pair (u, v) in G , concatenate u 's and v 's embeddings into a length- $2k$ vector; (ii) sample a training set of node pairs E'_{train}

Figure 5.3: Link prediction results vs. embedding dimensionality k (best viewed in color).

(with same size as E_{test}), such that half of the node pairs are from G' and the other half are node pairs not connected in G ; (iii) feed the length- $2k$ vectors of node pairs in E'_{train} into a logistic regression classifier; (iv) then use the classifier to obtain the scores of node pairs in E_{test} for link prediction. For VERSE and PBG, the inner product approach only works for undirected graphs, since VERSE and PBG generate only one embedding vector per node, due to which the inner product approach cannot differentiate (u, v) from (v, u) . Therefore, on directed graphs, we also employ the aforementioned edge features approach for VERSE and PBG.

Figure 5.3 shows AUC of each method when k varies from 16 to 256. NRP consistently outperforms all competitors, by a significant margin of up to 3% on *Orkut* and *Friendster*, and a large margin of 0.5% to 2% on other graphs. Compared with the best competitor, *i.e.*, AROPE, NRP achieves a considerable gain of 1.9% on *Orkut* when $k = 128$. Note that NRP outperforms all the PPR-based competitors, including Baseline, APP, VERSE and STRAP, over all datasets, which confirms the efficacy of our reweighting scheme in NRP,

Figure 5.4: Graph reconstruction results vs. K (best viewed in color).

and validates our analysis of the conventional PPR deficiency in Chapter 1. Moreover, we observe that VERSE is worse on directed graphs, *i.e.*, *Wiki* and *TWeibo*, although it is the best competitor on undirected graph *BlogCatalog*. This is because that VERSE generates only one embedding vector per node, making it fail to capture the asymmetric transitivity (*i.e.*, direction of edges) in directed graphs [108, 185], which is critical for link prediction. Our method, NRP, instead generates two embedding vectors per node and successfully distinguishes the edge directions and thus is more promising. STRAP and GA cannot efficiently handle large graphs (*i.e.*, *Youtube*, *TWeibo*, *Orkut*, *Twitter* and *Friendster*), since they require the materialization of a large $n \times n$ matrix, which is extremely costly in terms of both space and time; in contrast, NRP does not require to do so. NRP also consistently outperforms AROPE by about 2% absolute improvement on all graphs. For the other competitors, their performance is also less than satisfactory, as shown in the figures. In summary, for link prediction, NRP yields considerable performance improvements compared with the state-of-the-art methods, over graphs with various sizes.

Figure 5.5: Node classification results (best viewed in color).

5.4.3 Graph Reconstruction

Following previous work, for this task, we (i) take a set S of node pairs from the input graph G , (ii) compute the score of each pair using the same approach as in link prediction, and then (iii) examine the top- K node pairs to identify the fraction of them that correspond to the edges in G . This fraction is referred to as the *precision@K* of the method considered. On *Wiki* and *BlogCatalog*, we let S be the set of all possible node pairs. Meanwhile, on *Youtube* and *TWeibo*, following previous work [182, 183], we construct S by taking a 1% sample of the $\binom{n}{2}$ possible pairs of nodes. We exclude the results on *Orkut* and *Twitter* since 1% of all the possible node pairs from these two graphs are excessively large.

Figure 5.4 shows the performance of all methods for graph reconstruction, varying K from 10 to 10^6 . For readability, we split the results of each dataset into two sub-figures in vertical, and each sub-figure compares NRP against a subset of the competitors. NRP outperforms all competitors consistently on all datasets. NRP remains highly accurate when K increases to 10^4 or even 10^5 , while the precisions of other methods, especially GA, AROPE, RandNE, APP, VERSE and STRAP, drop significantly. Specifically, NRP achieves at least 90% precision when K reaches 10^4 on *Wiki*, *Blogcatalog* and *TWeibo*, which means at least 10% absolute improvement over state-of-the-art methods. In addition, on *Youtube*, NRP achieves 2-8% absolute improvement over the best competitors, including VERSE. The superiority of NRP over the other PPR-based methods, *i.e.*, Baseline, APP, VERSE and STRAP, in graph reconstruction demonstrates the power of our reweighting scheme. Meanwhile, the improvements over all other methods like GA, AROPE and RandNE implies that NRP accurately captures the structural information of the input graph via PPR.

5.4.4 Node Classification

Node classification aims to predict each node’s label(s) based on its embeddings. Following previous work [134], we first construct network embeddings from the input graph G , and use the embeddings and labels of a random subset of the nodes to train a one-vs-all logistic regression classifier, after which we test the classifier with the embeddings and labels of the remaining nodes. In particular, for **NRP**, **Baseline**, **APP**, **GA**, and **STRAP**, we first normalize the forward and backward vectors, respectively, of each node v , and then concatenate them as the feature representation of v before feeding it to the classifier. Note that the embeddings produced by **NRP** are weighted versions of that produced by **Baseline**, and thus, they have the same feature representation for each node v after the normalization, for the task of node classification.

Figure 5.5 shows the Micro-F1 score achieved by each method when the percentage of nodes used for training varies from 10% to 90% (*i.e.*, 0.1 to 0.9 in the figures). The Macro-F1 results are qualitatively similar and thus omitted for the interest of space. **NRP** consistently outperforms all competitors on *Wiki* and *TWeibo*, and has comparable performance to **ProNE** on *BlogCatalog* and *Youtube*. Specifically, on *Wiki*, **NRP** achieves an impressive improvement of at least 3% in Micro-F1 over existing methods and about 1% lead on *TWeibo*, which is considerable in contrast to that of our competitors. This demonstrates that **NRP** can accurately capture the graph structure via PPR. On *BlogCatalog* and *Youtube*, **NRP**, **NetHiex**, **VERSE** and **ProNE** all achieve comparable performance. **ProNE** is slightly better than **NRP**, but note that **ProNE** can only handle undirected graphs and is specifically designed for node classification task by employing graph spectrum and graph partition techniques. **NetHiex** also requires the input graphs to be undirected. **VERSE** cannot achieve the same high-quality performance on directed graphs (Figure 5.5(a) and 5.5(d)) as it does on undirected graphs (Figure 5.5(b) and 5.5(c)). The reason is that **VERSE** only generates one embedding vector per node, and neglects the directions of edges in the directed graphs, while our method **NRP** can preserve the directions. Typically, **NRP** achieves consistent and outstanding performance for node classification task over all the real-world graphs.

Figure 5.6: Running time vs. embedding dimensionality k (best viewed in color).

5.4.5 Efficiency

Figure 5.6 plots the time required by each method to construct embeddings, when k is varied from 16 to 256. Note that the Y-axis is in log-scale, and that the reported time excludes the overheads for loading datasets and outputting embeddings. We also omit any method with processing time exceeding 7 days. For a fair comparison, all methods are ran with a single thread.

Among all methods tested, NRP strikes the best balance between effectiveness and efficiency, and is up to 2 orders of magnitude faster than all methods except Baseline, ProNE, RandNE and AROPE. In addition, as illustrated in Figure 5.3, 5.4, and 5.5, RandNE and AROPE are both less effective compared to NRP for the three tasks. Furthermore, the results of RandNE, AROPE, and ProNE on directed graphs (*i.e.*, Wiki, TWeibo and Twitter) are all inferior to those of NRP, as shown in Figure 5.3, 5.4, and 5.5. This is because RandNE, AROPE, and ProNE are specifically designed for undirected graphs instead of

directed graphs. **ProNE** is also inferior to **NRP** in terms of link prediction and graph reconstruction on undirected graphs. Although **Baseline** runs faster than **NRP**, it is less effective than the latter in terms of link prediction and graph reconstruction, due to the deficiency of conventional PPR discussed in Chapter 1. Neither **GA** nor **STRAP** is able to scale to large graphs, which again manifests the power of our scalable PPR computation. The remaining methods either rely on expensive training phases (*e.g.*, **DeepWalk** and **VERSE**), or require constructing a huge matrix (*e.g.*, **NetSMF**), thereby failing to handle large graphs efficiently as well.

5.4.6 Parameter Analysis

We study the effect of varying the parameters in **NRP**, including α , ϵ , ℓ_1 and ℓ_2 , for link prediction on *Wiki*, *Blogcatalog* and *Youtube* datasets. Note that α is the decay factor in PPR (Equation (5.1) in Section 5.1); ϵ is the error threshold of **RandSVD** used in our PPR approximation (Algorithm 6); ℓ_1 is the number of iterations for computing PPR (Algorithm 6); ℓ_2 is the number of epochs for reweighting node embeddings (Algorithm 9). The AUC results are shown in Figure 5.7, when one of the parameters is varied, the others are kept as default values in Section 5.4.1.

Figure 5.7(a) displays the AUC of **NRP** when we vary the decay factor α from 0.1 to 0.9. As α increases, the performance downgrades since only limited local neighborhoods of nodes are preserved and high-order proximities are failed to be captured in the embeddings, which is consistent with the observation in [134, 172]. When $\alpha = 0.1$ or 0.2, the AUC score is the highest, which holds on all the three datasets. And thus our choice of $\alpha = 0.15$ makes sure that the best efficacy is achieved.

The AUC result of **NRP** when varying ϵ from 0.1 to 0.9 is depicted in Figure 5.7(b). According to Theorem 5.1, ϵ influences the accuracy of our PPR approximation. As shown in Figure 5.7(b), when ϵ is increased (*i.e.*, the error caused by **RandSVD** is larger), the AUC performance of the embedding decreases, especially on *Youtube* dataset. Therefore, we set ϵ to 0.2, which has the same excellent performance as 0.1 but is computationally more efficient.

In Figure 5.7(c), observe that the AUC of **NRP** grows significantly when we vary ℓ_1 from 1 to 15, and keeps stable and excellent for larger ℓ_1 from 15 to 40, which holds for

Figure 5.7: Link prediction results with varying parameters (best viewed in color).

Name	$ V $	$ E $	$ E_{old} $	$ E_{new} $	Type
VK	78.59K	5.35M	2.68M	2.67M	undirected
Digg	279.63K	1.73M	1.03M	701.59K	directed

Table 5.3: Dataset statistics ($K = 10^3$, $M = 10^6$).

all the three datasets. Recall that the accuracy of our PPR approximation is affected by ℓ_1 as well and when ℓ_1 increases, the approximate PPR scores are more accurate. According to Figure 5.7(c), our choice of $\ell_1 = 20$ is proper and robust.

Figure 5.7(d) shows the AUC of NRP when we vary ℓ_2 from 0 to 30. The AUC increases significantly when ℓ_2 is increased from 0 to 10, and then keeps stable for larger ℓ_2 values, which is consistent across the three datasets. When $\ell_2 = 0$, it is equivalent to disable our reweighting scheme and only use the conventional PPR for embedding, which is significantly inferior to the case that our NRP reweighting scheme is enabled, *e.g.*, when $\ell_2 = 10$. Specifically, on *Wiki*, the AUC is increased from 0.78 to 0.91 when ℓ_2 is varied from 0 to 10. This validates our insight about the drawback of vanilla PPR for embeddings and demonstrates the power of our proposed reweighting scheme. Further, Figure 5.7(d) also shows that our reweighting scheme converges quickly when the epochs are increased.

5.4.7 Additional Experiments

Link Prediction on Evolving Graphs. In this set of experiments, we evaluate the link performance of all methods on real-world datasets with real new links, *i.e.*, evolving

Figure 5.8: Link prediction performance on dynamic graphs (best viewed in color).

Figure 5.9: Scalability tests.

Figure 5.10: Running time with varying parameters (best viewed in color).

graphs. Table 5.3 shows the statistics of the datasets. Specifically, *VK* [134] and *Digg* [72] are two real-world social networks, where each node represents a user and a link represents the friendship or following relationship. For *VK*, $|E_{old}|$ denotes the social network snapshot of *VK* in 2016 and $|E_{new}|$ is the set of new links (*i.e.*, friendships)

in 2017. In terms of *Digg*, $|E_{old}|$ is the snapshot of the social network in 2008 and $|E_{new}|$ consists of new links (*i.e.*, following relationships) in 2009. We run all network embedding methods on $|E_{old}|$ and then employ the learned embeddings to predict the new links $|E_{new}|$. Figure 5.8 plots the AUC results of all methods on *VK* and *Digg*. It can be observed that **NRP** achieves similar performance as PPR-based methods **STRAP**, **VERSE** and **APP** on undirected graph *VK*. On directed graph *Digg*, **NRP** outperforms all competitors by at least a large margin of 0.7%. The experimental results indicate the effectiveness of **NRP** in predicting "real new links" on real-world datasets.

Scalability Tests. In this set of experiments, we verify the scalability of **NRP**. Following prior work [182, 183], we use synthetic graphs of different sizes generated by the Erdos Renyi random graph model [35]. We run **NRP** on these synthetic graphs with default parameter settings described in Section 5.4.1. We record the running time when fixing the number of nodes (as 10^6) or fixing the number of edges (as 10^7) while varying the other, *i.e.*, the number of edges in $\{2 \times 10^7, 4 \times 10^7, 6 \times 10^7, 8 \times 10^7, 1 \times 10^8\}$ and the number of nodes in $\{2 \times 10^5, 4 \times 10^5, 6 \times 10^5, 8 \times 10^5, 1 \times 10^6\}$, respectively. Figure 5.9(a) and 5.9(b) plot the running time of **NRP** when varying the number of nodes and the number of edges, respectively. It can be observed that the running time grows linearly with the number of nodes and the number of edges, respectively, confirming the time complexity of **NRP** as well as varying the scalability of **NRP**.

Running Time with Varying Parameters. Figures 5.10(a)-5.10(d) depict the results when varying ℓ_1 , ℓ_2 , α and ϵ on *Wiki*, *Blogcatalog*, *Youtube* and *Tweibo*, respectively. We can observe that the running time of **NRP** grows when we increase the values of ℓ_1 , ℓ_2 and ϵ but remain almost stable when increases α , which accords with the time complexity of **NRP**, *i.e.*, $O\left(\left(\frac{\log n}{\epsilon} + \ell_1\right)mk' + \frac{\log n}{\epsilon}nk'^2 + \ell_2nk'^2\right)$. Especially, Figure 5.10(b) shows that ℓ_2 has a greater impact on the running time compared with other parameters.

5.5 Summary

This chapter presents **NRP**, a novel, efficient and effective approach for homogeneous network embedding. **NRP** constructs embedding vectors based on personalized PageRank

values and reweights each node’s embedding vectors based on an objective function concerning the in-/out- degree of each node. We show that **NRP** runs in time almost linear to the size of the input graph, and that it requires less than four hours to process a graph with 1.2 billion edges. Extensive experiments on real data also demonstrate that **NRP** considerably outperforms the state of the arts in terms of the accuracy link prediction, graph reconstruction, and node classification tasks.

Chapter 6

Attributed Network Embedding via Node-Attribute PageRank¹

In this chapter, we consider the *attributed network embedding* problem. Given a graph G where each node is associated with a set of attributes, ANE maps each node $v \in G$ to a compact vector X_v , which can be used in downstream machine learning tasks. Ideally, X_v should capture node v 's *affinity* to each attribute, which considers not only v 's own attribute associations, but also those of its connected nodes along edges in G . It is challenging to obtain high-utility embeddings that enable accurate predictions; scaling effective ANE computation to massive graphs with millions of nodes pushes the difficulty of the problem to a whole new level. Existing solutions largely fail on such graphs, leading to prohibitive costs, low-quality embeddings, or both. We propose **PANE**, an effective and scalable approach to ANE computation for massive graphs that achieves state-of-the-art result quality on multiple benchmark datasets, measured by the accuracy of three common prediction tasks: attribute inference, link prediction, and node classification. In particular, for the large *MAG* data with over 59 million nodes, 0.98 billion edges, and 2000 attributes, **PANE** is the only known viable solution that obtains effective embeddings on a single server, within 12 hours. **PANE** obtains high scalability and effectiveness through three main algorithmic designs. First, it formulates the learning objective based on a novel *node-attribute PageRank* model for attributed networks. The resulting optimization task is still challenging on large graphs. Second, **PANE** includes a highly efficient solver for the above optimization problem, whose key module is

¹The work in this chapter has been published as [169] in PVLDB 2021.

a carefully designed initialization of the embeddings, which drastically reduces the number of iterations required to converge. Finally, PANE utilizes multi-core CPUs through non-trivial parallelization of the above solver, which achieves scalability while retaining the high quality of the resulting embeddings. Extensive experiments, comparing 10 existing approaches on 8 real datasets, demonstrate that PANE consistently outperforms all existing methods in terms of result quality, while being orders of magnitude faster.

This chapter is organized as follows. In Section 6.1, we formulate the ANE problem formally and present an overview of our solution. In Section 6.2 and Section 6.3, single-thread PANE and parallel PANE are introduced in detail, respectively. We evaluate PANE against all competitors in terms of effectiveness and efficiency in Section 6.4. Finally, Section 6.5 briefly summarizes this chapter. Table 6.1 lists the frequently used notations in this chapter.

6.1 Problem Formulation

This section first defines and analyzes the *attributed network embedding* problem. Then, an overview of the high-level idea and objective function of our solution is given.

6.1.1 Attributed Network Embedding

Let $G = (V, E_V, R, E_R)$ be an *attributed network* as defined in Definition 2.1. Note that for a categorical attribute such as marital status, we first apply a pre-processing step that transforms the attribute into a set of binary ones through one-hot encoding. Without loss of generality, we assume that G is a directed graph; if G is undirected, then we treat each edge (v_i, v_j) in G as a pair of directed edges with opposing directions: (v_i, v_j) and (v_j, v_i) . Given a space budget $k \ll n$, a *node embedding* maps a node $v \in V$ to a length- k vector. The general, hand-waving goal of attributed network embedding is to compute such an embedding X_v for each node v in the input graph, such that X_v captures the graph structure and attribute information surrounding node v . In addition, following previous work [101], we also allocate a space budget $\frac{k}{2}$ (explained later in Section 6.1.3) for each attribute $r \in R$, and aim to compute an *attribute embedding* vector for r of length $\frac{k}{2}$.

Table 6.1: Frequently used notations.

Notation	Description
$G=(V, E_V, R, E_R)$	A graph G with node set V , edge set E_V , attribute set R , and node-attribute association set E_R .
d	The numbers of attributes in G .
k	The space budget of embedding vectors.
$\mathbf{R}$	The attribute matrix of G .
$\mathbf{R}_r, \mathbf{R}_c$	The row-normalized and column-normalized attribute matrices. See Equation (6.1).
$\mathbf{F}, \mathbf{B}$	The forward and backward affinity matrices. See Equations (6.2) and (6.3).
$\mathbf{F}', \mathbf{B}'$	The approximate forward and backward affinity matrices. See Equation (6.7).
$\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}$	The forward and backward embedding vectors, and attribute embedding vectors.
n_b	The number of threads.

We define an attribute matrix $\mathbf{R} \in \mathbb{R}^{n \times d}$, such that $\mathbf{R}[v_i, r_j] = w_{i,j}$ is the weight associated with the entry $(v_i, r_j, w_{i,j}) \in E_R$. We refer to $\mathbf{R}[v_i]$ as node v_i 's *attribute vector*. Based on $\mathbf{R}$, we derive a row-normalized (resp. column-normalized) attribute matrices $\mathbf{R}_r$ (resp. $\mathbf{R}_c$) as follows:

$$\mathbf{R}_r[v_i, r_j] = \frac{\mathbf{R}[v_i, r_j]}{\sum_{r_l \in R} \mathbf{R}[v_i, r_l]}, \quad \mathbf{R}_c[v_i, r_j] = \frac{\mathbf{R}[v_i, r_j]}{\sum_{r_l \in R} \mathbf{R}[v_l, r_l]}. \quad (6.1)$$

Extended graph. Our solution utilizes an *extended graph* $\mathcal{G}$ that incorporates additional nodes and edges into G . To illustrate, Figure 6.1 shows an example extended graph $\mathcal{G}$ constructed based on an input attributed network G with 6 nodes v_1-v_6 and 3 attributes r_1-r_3 . The left part of the figure (in black) shows the topology of G , *i.e.*, the edge set E_V . The right part of the figure (in blue) shows $\mathcal{G}$'s representation of the attribute associations E_R in G . Specifically, for each attribute $r_j \in R$, we create an additional node in $\mathcal{G}$; then, For each entry in E_R , *e.g.*, $(v_3, r_1, w_{3,1})$, we include in $\mathcal{G}$ a pair of edges with opposing directions connecting the node (*e.g.*, v_3) with the corresponding attribute node (*e.g.*, r_1), with an edge weight (*e.g.*, $w_{3,1}$). Note that in this example, nodes v_1 and v_2 are not associated with any attribute.

Table 6.2: Targets for $\mathbf{X}[v_i] \cdot \mathbf{Y}[r_j]^\top$.

 Figure 6.1: Extended graph $\mathcal{G}$

	$\mathbf{Y}[r_1]$	$\mathbf{Y}[r_2]$	$\mathbf{Y}[r_3]$
$\mathbf{X}_f[v_1]$	1.0	0.92	0.47
$\mathbf{X}_b[v_1]$	0.93	0.88	1.17
$\mathbf{X}_f[v_2]$	1.0	0.92	0.47
$\mathbf{X}_b[v_2]$	1.11	1.08	0.8
$\mathbf{X}_f[v_3]$	1.12	1.04	0.54
$\mathbf{X}_b[v_3]$	1.06	0.95	0.99
$\mathbf{X}_f[v_5]$	0.98	1.1	1.08
$\mathbf{X}_b[v_5]$	1.09	1.22	0.61
$\mathbf{X}_f[v_6]$	0.89	0.82	2.05
$\mathbf{X}_b[v_6]$	0.53	0.61	1.6

6.1.2 Node-Attribute PageRank

As explained in Section 1.2 of Chapter 1, the resulting embedding of a node $v \in V$ should capture its *affinity* with attributes in R , where the affinity definition should take into account both the attributes directly associated with v in E_R , and the attributes of the nodes that v can reach via edges in E_V . To effectively model node-attribute affinity via multiple hops in $\mathcal{G}$, we employ an adaptation of the *personalized PageRank* [78, 132] technique to our setting with an extended graph $\mathcal{G}$. In the following, we refer to an PPR random walk simply as a *random walk*. Specifically, since $\mathcal{G}$ is directed, we distinguish two types of node-attribute affinity: *forward node-attribute PageRank*, denoted as $\mathbf{F}$, and *backward node-attribute PageRank*, denoted as $\mathbf{B}$.

Forward Node-Attribute PageRank. We first focus on forward node-attribute PageRank. Given an attributed graph G , a node v_i , and random walk stopping probability α ($0 < \alpha < 1$), a *forward random walk* on $\mathcal{G}$ starts from node v_i . At each step, assume that the walk is currently at node v_l . Then, the walk can either (i) with probability α , terminate at v_l , or (ii) with probability $1 - \alpha$, follow an edge in E_V to a random out-neighbor of v_l . After a random walk terminates at a node v_l , we randomly follow an edge in E_R to an attribute r_j , with probability $\mathbf{R}_r[v_l, r_j]$, i.e., a normalized edge weight defined in Equation (6.1)². The forward random walk yields a *node-to-attribute pair* (v_i, r_j) , and we add this pair to a collection $\mathcal{S}_f$.

²In the degenerate case that v_l is not associated with any attribute, e.g., v_1 in Figure 6.1, we simply restart the random walk from the source node v_i , and repeat the process.

Suppose that we sample n_r node-to-attribute pairs for each node v_i , the size of $\mathcal{S}_f$ is then $n_r \cdot n$, where n is the number of nodes in G . Denote $p_f(v_i, r_j)$ as the probability that a forward random walk starting from v_i yields a node-to-attribute pair (v_i, r_j) . Then, the *forward node-attribute PageRank* $\mathbf{F}[v_i, r_j]$ between node v_i and attribute r_j is defined as follows.

$$\mathbf{F}[v_i, r_j] = \log \left(\frac{n \cdot p_f(v_i, r_j)}{\sum_{v_h \in V} p_f(v_h, r_j)} + 1 \right) \quad (6.2)$$

To explain the intuition behind the above definition, note that in collection $\mathcal{S}_f$, the probabilities of observing node v_i , attribute r_j , and pair (v_i, r_j) are $\mathbb{P}(v_i) = \frac{1}{n}$, $\mathbb{P}(r_j) = \frac{\sum_{v_h \in V} p_f(v_h, r_j)}{n}$, and $\mathbb{P}(v_i, r_j) = \frac{p_f(v_i, r_j)}{n}$, respectively. Thus, the above definition of forward node-attribute PageRank is a variant of the PPR-based *pointwise mutual information* (PMI) [24] between node v_i and attribute r_j . In particular, given a collection of element pairs $\mathcal{S}$, the PMI of element pair $(x, y) \in \mathcal{S}$, denoted as $\text{PMI}(x, y)$, is defined as $\text{PMI}(x, y) = \log \left(\frac{\mathbb{P}(x, y)}{\mathbb{P}(x) \cdot \mathbb{P}(y)} \right)$, where $\mathbb{P}(x)$ (resp. $\mathbb{P}(y)$) is the probability of observing x (resp. y) in $\mathcal{S}$ and $\mathbb{P}(x, y)$ is the probability of observing pair (x, y) in $\mathcal{S}$. The larger $\text{PMI}(x, y)$ is, the more likely x and y co-occur in $\mathcal{S}$. Note that $\text{PMI}(x, y)$ can be negative. To avoid this, we use an alternative: shifted PMI, defined as $\text{SPMI}(x, y) = \log \left(\frac{\mathbb{P}(x, y)}{\mathbb{P}(x) \cdot \mathbb{P}(y)} + 1 \right)$, which is guaranteed to be positive, while retaining the original order of values of PMI. $\mathbf{F}[v_i, r_j]$ in Equation (6.2) is then $\text{SPMI}(v_i, r_j)$.

Another way to understand Equation (6.2) is through an analogy to TF/IDF [118] in natural language processing. Specifically, if we view the all forward random walks as a “document”, then $n \cdot p_f(v_i, r_j)$ is akin to the term frequency of r_j , whereas the denominator in Equation (6.2) is similar to the inverse document frequency of r_j . Thus, the normalization penalizes common attributes, and compensates for rare attributes.

Backward Node-Attribute PageRank. Next we define backward node-attribute PageRank in a similar fashion. Given an attributed network G , an attribute r_j and stopping probability α , a *backward random walk* starting from r_j first randomly samples a node v_l according to probability $\mathbf{R}_c[v_l, r_j]$, defined in Equation (6.1). Then, the walk starts from node v_l ; at each step, the walk either terminates at the current node with α probability, or randomly jumps to an out-neighbor of current node with $1 - \alpha$ probability. Suppose that the walk terminates at node v_i ; then, it returns an *attribute-to-node pair*

(r_j, v_i) , which is added to a collection $\mathcal{S}_b$. After sampling n_r attribute-to-node pairs for each attribute, the size of $\mathcal{S}_b$ becomes $n_r \cdot d$. Let $p_b(v_i, r_j)$ be the probability that a backward random walk starting from attribute r_j stops at node v_i . In collection $\mathcal{S}_b$, the probabilities of observing attribute r_j , node v_i and pair (r_j, v_i) are $\mathbb{P}(r_j) = \frac{1}{d}$, $\mathbb{P}(v_i) = \frac{\sum_{r_h \in R} p_b(v_i, r_h)}{d}$ and $\mathbb{P}(v_i, r_j) = \frac{p_b(v_i, r_j)}{d}$, respectively. We define backward node-attribute PageRank $\mathbf{B}[v_i, r_j]$ as follows.

$$\mathbf{B}[v_i, r_j] = \log \left(\frac{d \cdot p_b(v_i, r_j)}{\sum_{r_h \in R} p_b(v_i, r_h)} + 1 \right). \quad (6.3)$$

6.1.3 Solution Objective

Next we define our objective function for ANE, based on the notions of forward and backward node-attribute PageRank defined in Equation (6.2) and Equation (6.3), respectively. Let $\mathbf{F}[v_i, r_j]$ (resp. $\mathbf{B}[v_i, r_j]$) be the forward node-attribute PageRank (resp. backward node-attribute PageRank) between node v_i and attribute r_j . Given a space budget k , our objective is to learn (i) two embedding vectors for each node v_i , namely a *forward embedding vector*, denoted as $\mathbf{X}_f[v_i] \in \mathbb{R}^{\frac{k}{2}}$ and a *backward embedding vector*, denoted as $\mathbf{X}_b[v_i] \in \mathbb{R}^{\frac{k}{2}}$, as well as (ii) an *attribute embedding vector* $\mathbf{Y}[r_j] \in \mathbb{R}^{\frac{k}{2}}$ for each attribute r_j , such that the following objective is minimized:

$$\mathcal{O} = \min_{\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b} \sum_{v_i \in V} \sum_{r_j \in R} (\mathbf{F}[v_i, r_j] - \mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_j]^\top)^2 + (\mathbf{B}[v_i, r_j] - \mathbf{X}_b[v_i] \cdot \mathbf{Y}[r_j]^\top)^2 \quad (6.4)$$

Intuitively, in the above objective function, we approximate the forward node-attribute PageRank $\mathbf{F}[v_i, r_j]$ between node v_i and attribute r_j using the dot product of their respective embedding vectors, *i.e.*, $\mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_j]^\top$. Similarly, we also approximate the backward node-attribute PageRank using $\mathbf{X}_b[v_i] \cdot \mathbf{Y}[r_j]^\top$. The objective is then to minimize the total squared error of such approximations, over all nodes and all attributes in the input data.

Running Example. Assume that in the extended graph shown in Figure 6.1, all attribute weights in E_R are 1, and the random walk stopping probability α is set to default value 0.15 [78, 132]. Table 6.2 lists the target values, *i.e.*, the exact forward and backward

Algorithm 10: PANE (single thread)

Input: Attributed network G , space budget k , random walk stopping probability α , error threshold ϵ .

Output: Forward and backward embedding vectors $\mathbf{X}_f$, $\mathbf{X}_b$ and attribute embedding vectors $\mathbf{Y}$.

```

1  $t \leftarrow \frac{\log(\epsilon)}{\log(1-\alpha)} - 1$ ;
2  $\mathbf{F}', \mathbf{B}' \leftarrow \text{APMI}(\mathbf{P}, \mathbf{R}, \alpha, t)$ ;
3  $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b \leftarrow \text{SVDCCD}(\mathbf{F}', \mathbf{B}', k, t)$ ;
4 return  $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$ ;
    
```

node-attribute PageRank values. According to Equation (6.4), for the inner products of attribute embedding vectors of r_1 - r_3 and embedding vectors of v_1 - v_6 . These values are calculated based on Equations (6.2) and (6.3), using simulated random walks on the extended graph in Figure 6.1. Observe, for example, that node v_1 has high node-attribute PageRank values (both forward and backward) with attribute r_1 , which agrees with the intuition that v_1 is connected to r_1 via many different intermediate nodes, *i.e.*, v_3, v_4, v_5 . Considering v_5 , we observe that it has higher forward affinity value with r_3 than that with r_1 , which violates the fact that v_5 owns r_1 but r_3 . If we take into account both forward and backward affinity values, the problem is resolved. This could be utilized for attribute inference, as is done in our experiments in Section 6.4.2.

Our problem formulation involves rather expensive random walks in the computation of the affinity matrices (*i.e.*, $\mathbf{F}$ and $\mathbf{B}$). It is challenging to compute approximate forward and backward affinity for every node-attribute pair in the input graph both efficiently and accurately. Unlike prior work, which typically preserves one type of node-node affinity into embeddings, we aim to capture both forward and backward affinity in the embeddings by solving the optimization problem in Equation (6.4)), which is much harder to converge. Additionally, it disables us to employ efficient matrix factorization techniques over the affinity matrices to obtain embeddings straightforwardly. Next we present the proposed algorithm PANE, which is capable of solving the problem on massive graphs.

6.2 The PANE Algorithm

For ease of presentation, this section describes the single-threaded version of the proposed solution PANE for ANE. The full version of PANE that runs in multiple threads is

Algorithm 11: APMI

Input: $\mathbf{P}, \mathbf{R}, \alpha, t$.

Output: $\mathbf{F}', \mathbf{B}'$.

- 1 Compute $\mathbf{R}_r$ and $\mathbf{R}_c$ by Equation (6.1);
 - 2 $\mathbf{P}_f^{(0)} \leftarrow \mathbf{R}_r, \mathbf{P}_b^{(0)} \leftarrow \mathbf{R}_c$;
 - 3 **for** $\ell \leftarrow 1$ **to** t **do**
 - 4 $\mathbf{P}_f^{(\ell)} \leftarrow (1 - \alpha) \cdot \mathbf{P} \mathbf{P}_f^{(\ell-1)} + \alpha \cdot \mathbf{R}_r$;
 - 5 $\mathbf{P}_b^{(\ell)} \leftarrow (1 - \alpha) \cdot \mathbf{P}^\top \mathbf{P}_b^{(\ell-1)} + \alpha \cdot \mathbf{R}_c$;
 - 6 Normalize $\mathbf{P}_f^{(t)}$ by columns to get $\widehat{\mathbf{P}}_f^{(t)}$;
 - 7 Normalize $\mathbf{P}_b^{(t)}$ by rows to get $\widehat{\mathbf{P}}_b^{(t)}$;
 - 8 $\mathbf{F}' \leftarrow \log(n \cdot \widehat{\mathbf{P}}_f^{(t)} + 1)$;
 - 9 $\mathbf{B}' \leftarrow \log(d \cdot \widehat{\mathbf{P}}_b^{(t)} + 1)$;
 - 10 **return** $\mathbf{F}', \mathbf{B}'$;
-

explained later in Section 6.3. At a high level, PANE consists of two phases: (i) iteratively computing approximated versions $\mathbf{F}'$ and $\mathbf{B}'$ of the forward and backward node-attribute PageRank matrices with rigorous approximation error guarantees (Section 6.2.1), and (ii) initializing the embedding vectors with a greedy algorithm, and then jointly factorizing $\mathbf{F}'$ and $\mathbf{B}'$ using *cyclic coordinate descent* [156] to obtain the output embedding vectors $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$ (Section 6.2.2). Given an input attributed network G , space budget k , random walk stopping probability α and an error threshold ϵ as inputs, Algorithm 10 outlines the proposed PANE algorithm in the single-threaded setting.

6.2.1 Forward and Backward Node-Attribute PageRank Approximation

In Section 6.1, node-attribute PageRank values are defined using a large number of random walks, which are expensive to simulate on a massive graph. To tackle this, we transform forward and backward PageRank in Equations (6.2) and (6.3) into *iterative matrix computations*, and propose APMI in Algorithm 11 to efficiently approximate matrices $\mathbf{F}$ and $\mathbf{B}$ with error guarantees, without sampling random walks.

Observe that in Equations (6.2) and (6.3), the key for forward and backward PageRank computation is to obtain $p_f(v_i, r_j)$ and $p_b(v_i, r_j)$ for every pair $(v_i, r_j) \in V \times R$. Recall that $p_f(v_i, r_j)$ is the probability that a forward random walk starting from node v_i picks

attribute r_j , while $p_b(v_i, r_j)$ is the probability of a backward random walk from attribute r_j stopping at node v_i . Given nodes v_i and v_l , denote $\pi(v_i, v_l)$ as the probability that a random walk starting from v_i stops at v_l , *i.e.*, the random walk score of v_l with respect to v_i . By definition, $p_f(v_i, r_j) = \sum_{v_l \in V} \pi(v_i, v_l) \cdot \mathbf{R}_r[v_l, r_j]$, where $\mathbf{R}_r[v_l, r_j]$ is the probability that node v_l picks attribute r_j , according to Equation (6.1). Similarly, $p_b(v_i, r_j)$ is formulated as $p_b(v_i, r_j) = \sum_{v_l \in V} \mathbf{R}_c[v_l, r_j] \cdot \pi(v_l, v_i)$, where $\mathbf{R}_c[v_l, r_j]$ is the probability that attribute r_j picks node v_l from all nodes having r_j based on their attribute weights. By the definition of random walk scores in [78, 132], we can derive the matrix form of p_f and p_b as follows.

$$\begin{aligned} \mathbf{P}_f &= \alpha \sum_{\ell=0}^{\infty} (1 - \alpha)^\ell \mathbf{P}^\ell \cdot \mathbf{R}_r, \\ \mathbf{P}_b &= \alpha \sum_{\ell=0}^{\infty} (1 - \alpha)^\ell \mathbf{P}^{\top \ell} \cdot \mathbf{R}_c. \end{aligned} \quad (6.5)$$

We only consider t iterations to approximate $\mathbf{P}_f$ and $\mathbf{P}_b$ in Equation (6.6), where t is set to $\frac{\log(\epsilon)}{\log(1-\alpha)} - 1$.

$$\begin{aligned} \mathbf{P}_f^{(t)} &= \alpha \sum_{\ell=0}^t (1 - \alpha)^\ell \mathbf{P}^\ell \cdot \mathbf{R}_r, \\ \mathbf{P}_b^{(t)} &= \alpha \sum_{\ell=0}^t (1 - \alpha)^\ell \mathbf{P}^{\top \ell} \cdot \mathbf{R}_c. \end{aligned} \quad (6.6)$$

Then, we normalize $\mathbf{P}_f^{(t)}$ by columns and $\mathbf{P}_b^{(t)}$ by rows as follows.

$$\widehat{\mathbf{P}}_f^{(t)}[v_i, r_j] = \frac{\mathbf{P}_f^{(t)}[v_i, r_j]}{\sum_{v_l \in V} \mathbf{P}_f^{(t)}[v_l, r_j]}, \quad \widehat{\mathbf{P}}_b^{(t)}[v_i, r_j] = \frac{\mathbf{P}_b^{(t)}[v_i, r_j]}{\sum_{r_l \in R} \mathbf{P}_b^{(t)}[v_i, r_l]}$$

After normalization, we compute $\mathbf{F}'$ and $\mathbf{B}'$ according to the definitions of forward and backward PageRank as follows.

$$\mathbf{F}' = \log(n \cdot \widehat{\mathbf{P}}_f^{(t)} + 1), \quad \mathbf{B}' = \log(d \cdot \widehat{\mathbf{P}}_b^{(t)} + 1) \quad (6.7)$$

Algorithm 11 shows the pseudo-code of APMI to compute $\mathbf{F}'$ and $\mathbf{B}'$. Specifically, APMI takes as inputs random walk matrix $\mathbf{P}$, attribute matrix $\mathbf{R}$, random walk stopping

probability α and the number of iterations t . At Line 1, **APMI** begins by computing row-normalized attribute matrix $\mathbf{R}_r$ and column-normalized attribute matrix $\mathbf{R}_c$ according to Equation (6.1). Then, **APMI** computes $\mathbf{P}_f^{(t)}$ and $\mathbf{P}_b^{(t)}$ based on Equation (6.6). Note that $\mathbf{P}$ is sparse and has m non-zero entries. Thus, the computations of $\alpha \sum_{\ell=0}^t (1-\alpha)^\ell \mathbf{P}^\ell$ and $\alpha \sum_{\ell=0}^t (1-\alpha)^\ell \mathbf{P}^\top{}^\ell$ in Equation (6.6) need $O(mnt)$ time, which is prohibitively expensive on large graphs. We avoid such expensive overheads and achieve a time cost of $O(mdt)$ for computing $\mathbf{P}_f^{(t)}$ and $\mathbf{P}_b^{(t)}$ by an iterative process as follows. Initially, we set $\mathbf{P}_f^{(0)} = \mathbf{R}_r$ and $\mathbf{P}_b^{(0)} = \mathbf{R}_c$ (Line 2). Then, we start an iterative process from Line 3 to 5 with t iterations; at the ℓ -th iteration, we compute $\mathbf{P}_f^{(\ell)} = (1-\alpha) \cdot \mathbf{P} \mathbf{P}_f^{(\ell-1)} + \alpha \cdot \mathbf{R}_r$ and $\mathbf{P}_b^{(\ell)} = (1-\alpha) \cdot \mathbf{P}^\top \mathbf{P}_b^{(\ell-1)} + \alpha \cdot \mathbf{R}_c$. After t iterations, **APMI** normalizes $\mathbf{P}_f^{(t)}$ by column and $\mathbf{P}_b^{(t)}$ by row (Lines 6-7). At Lines 8-9, **APMI** obtains $\mathbf{F}'$ and $\mathbf{B}'$ as the approximate forward and backward PageRank matrices.

Lemma 6.1. *Given $\mathbf{P}, \mathbf{R}_r, \alpha, \epsilon$ as inputs to Algorithm 11, the returned approximate forward and backward PageRank matrices $\mathbf{F}', \mathbf{B}'$ satisfy that, for every pair $(v_i, r_j) \in V \times R$,*

$$\begin{aligned} \frac{2^{\mathbf{F}'[v_i, r_j]} - 1}{2^{\mathbf{F}[v_i, r_j]} - 1} &\in \left[\max \left\{ 0, 1 - \frac{\epsilon}{\mathbf{P}_f[v_i, r_j]} \right\}, 1 + \frac{\epsilon}{\sum_{v_l \in V} \max \{0, \mathbf{P}_f[v_l, r_j] - \epsilon\}} \right], \\ \frac{2^{\mathbf{B}'[v_i, r_j]} - 1}{2^{\mathbf{B}[v_i, r_j]} - 1} &\in \left[\max \left\{ 0, 1 - \frac{\epsilon}{\mathbf{P}_b[v_i, r_j]} \right\}, 1 + \frac{\epsilon}{\sum_{r_l \in R} \max \{0, \mathbf{P}_b[v_i, r_l] - \epsilon\}} \right]. \end{aligned}$$

Proof. First, with $t = \frac{\log(\epsilon)}{\log(1-\alpha)} - 1$, we have

$$\sum_{\ell=t+1}^{\infty} \alpha(1-\alpha)^\ell = 1 - \sum_{\ell=0}^t \alpha(1-\alpha)^\ell = (1-\alpha)^{t+1} = \epsilon. \quad (6.8)$$

By the definitions of $\mathbf{P}_f, \mathbf{P}_f^{(t)}$ and $\mathbf{P}_b, \mathbf{P}_b^{(t)}$ (i.e., Equation (6.5) and Equation (6.6)), for

every pair $(v_i, r_j) \in V \times R$,

$$\begin{aligned}
 \mathbf{P}_f[v_i, r_j] - \mathbf{P}_f^{(t)}[v_i, r_j] &= \sum_{\ell=t+1}^{\infty} \alpha(1-\alpha)^\ell \mathbf{P}^\ell[v_i] \cdot \mathbf{R}_r^\top[r_j] \\
 &= \left(\sum_{\ell=t+1}^{\infty} \alpha(1-\alpha)^\ell \mathbf{P}^\ell \right) [v_i] \cdot \mathbf{R}_r^\top[r_j] \leq \sum_{\ell=t+1}^{\infty} \alpha(1-\alpha)^\ell = \epsilon, \\
 \mathbf{P}_b[v_i, r_j] - \mathbf{P}_b^{(t)}[v_i, r_j] &= \sum_{\ell=t+1}^{\infty} \alpha(1-\alpha)^\ell \mathbf{P}^{\top\ell}[v_i] \cdot \mathbf{R}_c^\top[r_j] \\
 &\leq \sum_{v_l \in V} \left(\sum_{\ell=t+1}^{\infty} \alpha(1-\alpha)^\ell \right) \cdot \mathbf{R}_c[v_l, r_j] \leq \sum_{v_l \in V} \epsilon \cdot \mathbf{R}_c[v_l, r_j] = \epsilon.
 \end{aligned}$$

Based on the above inequalities, for every pair $(v_i, r_j) \in V \times R$,

$$\max\{0, \mathbf{P}_f[v_i, r_j] - \epsilon\} \leq \mathbf{P}_f^{(t)}[v_i, r_j] \leq \mathbf{P}_f[v_i, r_j], \quad (6.9)$$

$$\max\{0, \mathbf{P}_b[v_i, r_j] - \epsilon\} \leq \mathbf{P}_b^{(t)}[v_i, r_j] \leq \mathbf{P}_b[v_i, r_j]. \quad (6.10)$$

According to Lines 6-9 in Algorithm 11, for every pair $(v_i, r_j) \in V \times R$,

$$\frac{2^{\mathbf{F}'[v_i, r_j]} - 1}{2^{\mathbf{F}[v_i, r_j]} - 1} = \frac{\widehat{\mathbf{P}}_f^{(t)}[v_i, r_j]}{\widehat{\mathbf{P}}_f[v_i, r_j]} = \frac{\mathbf{P}_f^{(t)}[v_i, r_j]}{\sum_{v_l \in V} \mathbf{P}_f^{(t)}[v_l, r_j]} \times \frac{\sum_{v_l \in V} \mathbf{P}_f[v_l, r_j]}{\mathbf{P}_f[v_i, r_j]}, \quad (6.11)$$

$$\frac{2^{\mathbf{B}'[v_i, r_j]} - 1}{2^{\mathbf{B}[v_i, r_j]} - 1} = \frac{\widehat{\mathbf{P}}_b^{(t)}[v_i, r_j]}{\widehat{\mathbf{P}}_b[v_i, r_j]} = \frac{\mathbf{P}_b^{(t)}[v_i, r_j]}{\sum_{r_l \in R} \mathbf{P}_b^{(t)}[v_i, r_l]} \times \frac{\sum_{r_l \in R} \mathbf{P}_b[v_i, r_l]}{\mathbf{P}_b[v_i, r_j]}. \quad (6.12)$$

Plugging Inequalities (6.9) and (6.10) into Inequalities (6.11) and (6.12) leads to the desired results, which completes our proof. $\square$

6.2.2 Joint Factorization of Node-Attribute PageRank Matrices

This section presents the proposed algorithm SVDCCD, outlined in Algorithm 13, which jointly factorizes the approximate forward and backward PageRank matrices $\mathbf{F}'$ and $\mathbf{B}'$, in order to obtain the embedding vectors of all nodes and attributes, *i.e.*, $\mathbf{X}_f$, $\mathbf{X}_b$, and $\mathbf{Y}$. As the name suggests, the proposed SVDCCD solver is based on the *cyclic coordinate descent (CCD)* framework, which iteratively updates each embedding value towards optimizing the objective function in Equation (6.4). The problem, however, is

that a direct application of CCD, starting from random initial values of the embeddings, requires numerous iterations to converge, leading to prohibitive computational costs. We address this problem with a greedy initialization method, as explained in the following.

Greedy initialization. In many optimization problems, all we need for efficiency is a good initialization. Thus, a key component in the proposed SVDCCD algorithm is such an initialization of embedding values, based on *singular value decomposition* (SVD) [56]. Note that unlike other matrix factorization problems, here SVD by itself cannot solve our problem, because the objective function in Equation (6.4) requires the joint factorization of both the forward and backward PageRank matrices at the same time, which cannot be directly addressed with SVD.

Algorithm 12 describes the proposed GreedyInit module of SVDCCD, which initializes embeddings $\mathbf{X}_f$, $\mathbf{X}_b$, and $\mathbf{Y}$. Specifically, the algorithm first employs an efficient randomized SVD algorithm [104] at Line 1 to decompose $\mathbf{F}'$ into $\mathbf{U} \in \mathbb{R}^{n \times \frac{k}{2}}$, $\mathbf{\Sigma} \in \mathbb{R}^{\frac{k}{2} \times \frac{k}{2}}$, $\mathbf{V} \in \mathbb{R}^{d \times \frac{k}{2}}$, and then initializes $\mathbf{X}_f = \mathbf{U}\mathbf{\Sigma}$ and $\mathbf{Y} = \mathbf{V}$ at Line 2, which satisfies $\mathbf{X}_f \cdot \mathbf{Y}^\top \approx \mathbf{F}'$. In other words, this initialization immediately gains a good approximation of the forward PageRank matrix.

Recall that our objective function in Equation (6.4) also aims to find $\mathbf{X}_b$ such that $\mathbf{X}_b \mathbf{Y}^\top \approx \mathbf{B}'$, *i.e.*, to approximate the backward PageRank matrix well. Here comes the key observation of the algorithm: that matrix $\mathbf{V}$ (*i.e.*, $\mathbf{Y}$) returned by exact SVD is *unitary*, *i.e.*, $\mathbf{Y}^\top \mathbf{Y} = \mathbf{I}$, which implies that $\mathbf{X}_b \approx \mathbf{X}_b \mathbf{Y}^\top \mathbf{Y} \approx \mathbf{B}' \mathbf{Y}$. Accordingly, we seed $\mathbf{X}_b$ with $\mathbf{B}' \mathbf{Y}$ at Line 2 of Algorithm 12. This initialization of $\mathbf{X}_b$ also leads to a relatively good approximation of the backward PageRank matrix. Consequently, the number of iterations required by SVDCCD is drastically reduced, as confirmed by our experiments in Section 6.4.

Efficient refinement of the initial embeddings. In Algorithm 13, after initializing $\mathbf{X}_f$, $\mathbf{X}_b$ and $\mathbf{Y}$ at Line 1, we apply cyclic coordinate descent to refine the embedding vectors according to our objective function in Equation (6.4) from Lines 2 to 14. The basic idea of CCD is to cyclically iterate through all entries in $\mathbf{X}_f$, $\mathbf{X}_b$ and $\mathbf{Y}$, one at a time, minimizing the objective function with respect to each entry (*i.e.*, coordinate direction). Specifically, in each iteration, CCD updates each entry of $\mathbf{X}_f$, $\mathbf{X}_b$ and $\mathbf{Y}$

according to the following rules:

$$\mathbf{X}_f[v_i, l] \leftarrow \mathbf{X}_f[v_i, l] - \mu_f(v_i, l), \quad (6.13)$$

$$\mathbf{X}_b[v_i, l] \leftarrow \mathbf{X}_b[v_i, l] - \mu_b(v_i, l), \quad (6.14)$$

$$\mathbf{Y}[r_j, l] \leftarrow \mathbf{Y}[r_j, l] - \mu_y(r_j, l), \quad (6.15)$$

with $\mu_f(v_i, l)$, $\mu_b(v_i, l)$ and $\mu_y(r_j, l)$ computed by:

$$\mu_f(v_i, l) = \frac{\mathbf{S}_f[v_i] \cdot \mathbf{Y}[:, l]}{\mathbf{Y}^\top[l] \cdot \mathbf{Y}[:, l]}, \quad \mu_b(v_i, l) = \frac{\mathbf{S}_b[v_i] \cdot \mathbf{Y}[:, l]}{\mathbf{Y}^\top[l] \cdot \mathbf{Y}[:, l]}, \quad (6.16)$$

$$\mu_y(r_j, l) = \frac{\mathbf{X}_f^\top[l] \cdot \mathbf{S}_f[:, r_j] + \mathbf{X}_b^\top[l] \cdot \mathbf{S}_b[:, r_j]}{\mathbf{X}_f^\top[l] \cdot \mathbf{X}_f[:, l] + \mathbf{X}_b^\top[l] \cdot \mathbf{X}_b[:, l]}, \quad (6.17)$$

where $\mathbf{S}_f = \mathbf{X}_f \mathbf{Y}^\top - \mathbf{F}'$ and $\mathbf{S}_b = \mathbf{X}_b \mathbf{Y}^\top - \mathbf{B}'$ are obtained at Line 3 in Algorithm 12.

However, directly applying the above updating rules to learn $\mathbf{X}_f$, $\mathbf{X}_b$, and $\mathbf{Y}$ is inefficient, leading to many redundant matrix operations. Lines 2-14 in Algorithm 13 show how to efficiently apply the above updating rules by dynamically maintaining and partially updating intermediate results. Specifically, each iteration in Lines 3-14 first fixes $\mathbf{Y}$ and updates each row of $\mathbf{X}_f$ and $\mathbf{X}_b$ (Lines 3-9), and then updates each column of $\mathbf{Y}$ with $\mathbf{X}_f$ and $\mathbf{X}_b$ fixed (Lines 10-14). According to Equations (6.16) and (6.17), $\mu_f(v_i, l)$, $\mu_b(v_i, l)$, and $\mu_y(r_j, l)$ are pertinent to $\mathbf{S}_f[v_i]$, $\mathbf{S}_b[v_i]$, and $\mathbf{S}_f[:, r_j]$, $\mathbf{S}_b[:, r_j]$ respectively, where $\mathbf{S}_f$ and $\mathbf{S}_b$ further depend on embedding vectors $\mathbf{X}_f$, $\mathbf{X}_b$ and $\mathbf{Y}$. Therefore, whenever $\mathbf{X}_f[v_i, l]$, $\mathbf{X}_b[v_i, l]$, and $\mathbf{Y}[r_j, l]$ are updated in the iteration (Lines 6-7 and Line 13), $\mathbf{S}_f$ and $\mathbf{S}_b$ need to be updated accordingly. Directly recomputing $\mathbf{S}_f$ and $\mathbf{S}_b$ by $\mathbf{S}_f = \mathbf{X}_f \mathbf{Y}^\top - \mathbf{F}'$ and $\mathbf{S}_b = \mathbf{X}_b \mathbf{Y}^\top - \mathbf{B}'$ whenever an entry in $\mathbf{X}_f$, $\mathbf{X}_b$ and, $\mathbf{Y}$ is updated is expensive.

Instead, we dynamically maintain and partially update $\mathbf{S}_f$ and $\mathbf{S}_b$ according to Equations (6.18), (6.19) and (6.20). Specifically, when $\mathbf{X}_f[v_i, l]$ and $\mathbf{X}_b[v_i, l]$ are updated (Lines 6-7), we update $\mathbf{S}_f[v_i]$ and $\mathbf{S}_b[v_i]$ respectively with $O(d)$ time at Lines 8-9 by

$$\mathbf{S}_f[v_i] \leftarrow \mathbf{S}_f[v_i] - \mu_f(v_i, l) \cdot \mathbf{Y}[:, l]^\top, \quad (6.18)$$

$$\mathbf{S}_b[v_i] \leftarrow \mathbf{S}_b[v_i] - \mu_b(v_i, l) \cdot \mathbf{Y}[:, l]^\top, \quad (6.19)$$

Algorithm 12: GreedyInit

Input: $\mathbf{F}', \mathbf{B}', k, t$.
Output: $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b$.
 1 $\mathbf{U}, \Sigma, \mathbf{V} \leftarrow \text{RandSVD}(\mathbf{F}', \frac{k}{2}, t)$;
 2 $\mathbf{Y} \leftarrow \mathbf{V}, \mathbf{X}_f \leftarrow \mathbf{U}\Sigma, \mathbf{X}_b \leftarrow \mathbf{B}' \cdot \mathbf{Y}$;
 3 $\mathbf{S}_f \leftarrow \mathbf{X}_f \mathbf{Y}^\top - \mathbf{F}', \mathbf{S}_b \leftarrow \mathbf{X}_b \mathbf{Y}^\top - \mathbf{B}'$;
 4 **return** $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b$;

Algorithm 13: SVDCCD

Input: $\mathbf{F}', \mathbf{B}', k, t$.
Output: $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$.
 1 $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b \leftarrow \text{GreedyInit}(\mathbf{F}', \mathbf{B}', k, t)$;
 2 **for** $\ell \leftarrow 1$ **to** t **do**
 3 **for** $v_i \in V$ **do**
 4 **for** $l \leftarrow 1$ **to** $\frac{k}{2}$ **do**
 5 Compute $\mu_f(v_i, l), \mu_b(v_i, l)$ by Equation (6.16);
 6 $\mathbf{X}_f[v_i, l] \leftarrow \mathbf{X}_f[v_i, l] - \mu_f(v_i, l)$;
 7 $\mathbf{X}_b[v_i, l] \leftarrow \mathbf{X}_b[v_i, l] - \mu_b(v_i, l)$;
 8 Update $\mathbf{S}_f[v_i]$ by Equation (6.18);
 9 Update $\mathbf{S}_b[v_i]$ by Equation (6.19);
 10 **for** $r_j \in R$ **do**
 11 **for** $l \leftarrow 1$ **to** $\frac{k}{2}$ **do**
 12 Compute $\mu_y(r_j, l)$ by Equation (6.17);
 13 $\mathbf{Y}[r_j, l] \leftarrow \mathbf{Y}[r_j, l] - \mu_y(r_j, l)$;
 14 Update $\mathbf{S}_f[:, r_j], \mathbf{S}_b[:, r_j]$ by Equation (6.20);
 15 **return** $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$;

Whenever $\mathbf{Y}[r_j, l]$ is updated at Line 13, both $\mathbf{S}_f[:, r_j]$ and $\mathbf{S}_b[:, r_j]$ are updated in $O(n)$ time at Line 14 by

$$\begin{aligned}
 \mathbf{S}_f[:, r_j] &\leftarrow \mathbf{S}_f[:, r_j] - \mu_y(r_j, l) \cdot \mathbf{X}_f[:, l], \\
 \mathbf{S}_b[:, r_j] &\leftarrow \mathbf{S}_b[:, r_j] - \mu_y(r_j, l) \cdot \mathbf{X}_b[:, l],
 \end{aligned} \tag{6.20}$$

6.2.3 Complexity Analysis

In the proposed algorithm PANE (Algorithm 10), the maximum length of random walk is $t = \frac{\log(\epsilon)}{\log(1-\alpha)} - 1 = \frac{\log(\frac{1}{\epsilon})}{\log(\frac{1}{1-\alpha})} - 1$. According to Section 6.2.1, Algorithm 11 runs in time $O(md \cdot t) = O\left(md \cdot \log \frac{1}{\epsilon}\right)$. Meanwhile, according to [104], given $\mathbf{F}' \in \mathbb{R}^{n \times d}$ as input,

Algorithm 14: PANE

Input: Attributed network G , space budget k , random walk stopping probability α , error threshold ϵ , the number of threads n_b .

Output: Forward and backward embedding vectors $\mathbf{X}_f$, $\mathbf{X}_b$ and attribute embedding vectors $\mathbf{Y}$.

- 1 Partition V into n_b subsets $\mathcal{V} \leftarrow \{V_1, \dots, V_{n_b}\}$ equally;
 - 2 Partition R into n_b subsets $\mathcal{R} \leftarrow \{R_1, \dots, R_{n_b}\}$ equally;
 - 3 $t \leftarrow \frac{\log(\epsilon)}{\log(1-\alpha)} - 1$;
 - 4 $\mathbf{F}', \mathbf{B}' \leftarrow \text{PAPMI}(\mathbf{P}, \mathbf{R}, \alpha, t, \mathcal{V}, \mathcal{R})$;
 - 5 $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b \leftarrow \text{PSVDCCD}(\mathbf{F}', \mathbf{B}', \mathcal{V}, \mathcal{R}, k, t)$;
 - 6 **return** $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$;
-

RandSVD in Algorithm 12 requires $O(ndkt)$ time, where n , d , k are the number of nodes, number of attributes, and embedding space budget, respectively. The computation of $\mathbf{S}_f, \mathbf{S}_b$ costs $O(ndk)$ time. In addition, the t iterations of CCD for updating $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$ take $O(ndkt) = O(ndk \log \frac{1}{\epsilon})$ time. Therefore, the overall time complexity of Algorithm 10 is

$$O\left((md + ndk) \cdot \log\left(\frac{1}{\epsilon}\right)\right).$$

The memory consumption of intermediate results yielded in Algorithm 10, *i.e.*, $\mathbf{F}', \mathbf{B}'$, $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}, \mathbf{S}_f, \mathbf{S}_b$ are at most $O(nd)$. Therefore, the space complexity of Algorithm 10 is bounded by $O(nd + m)$, where the m term is due to the space required by the input graph.

6.3 Parallelization of PANE

Although single-thread PANE (*i.e.*, Algorithm 10) runs in linear time to the size of the input attributed network, it still requires substantial time to handle large-scale attributed networks in practice. For instance, on *MAG* dataset that has 59.3 million nodes, PANE (single thread) costs about five days. To further boost efficiency, we develop a parallel PANE (Algorithm 14) in this section, and it costs only 11.9 hours on *MAG* when using 10 threads (*i.e.*, up to 10 times speedup with 10 threads). However, it is challenging to develop a parallel algorithm achieving such linear scalability to the number of threads on multi-core CPU. Specifically, PANE involves different computational patterns, including intensive matrix computation, factorization, and cyclic coordinate descent updates.

Therefore, it is non-trivial to assign computing tasks of both nodes and attributes to threads, to fully utilize the parallel power. Moreover, it is also challenging to maintain the intermediate result of each thread and combine them as final result. To address the challenges, we propose several non-trivial parallelization techniques for our parallel PANE. In the first phase, we adopt block matrix multiplication [57] and propose PAPMI to compute forward and backward affinity matrices in a parallel manner (Section 6.3.1). In the second phase, we develop PSVDCCD with a split-merge-based parallel SVD technique to efficiently decompose affinity matrices, and further we propose a parallel CCD technique to refine the embedding vectors efficiently (Section 6.3.2).

Algorithm 14 illustrates the pseudo-code of parallel PANE. Compared to the single-thread version, parallel PANE takes as input an additional parameter, the number of threads n_b , and partitions the node set V and attribute set R into n_b subsets with equal size, denoted as $\mathcal{V}$ and $\mathcal{R}$, respectively (Lines 1-2). PANE invokes PAPMI (Algorithm 15) at Line 4 to derive $\mathbf{F}'$ and $\mathbf{B}'$, and then invokes PSVDCCD (Algorithm 17) to refine the embedding vectors.

Note that the parallel version of PANE does not return exactly the same outputs as the single-thread version, as some modules (*e.g.*, the parallel version of SVD) introduce additional error. Nevertheless, as the experiments in Section 6.4 demonstrates, the degradation of result utility in parallel PANE is small, when running on a moderate number of CPU cores (10 in our experiments), which is well compensated by the running time reduction of parallel PANE.

6.3.1 Parallel Forward and Backward Node-Attribute PageRank Approximation

To utilize multiple simultaneous threads, we employ block matrix multiplication [57] and propose PAPMI in Algorithm 15 to estimate $\mathbf{F}'$ and $\mathbf{B}'$ efficiently in parallel. In particular, after obtaining $\mathbf{R}_r$ and $\mathbf{R}_c$ based on Equation (6.1) at Line 1, PAPMI divides $\mathbf{R}_r$ and $\mathbf{R}_c$ into matrix blocks according to two input parameters, the node subsets $\mathcal{V} = \{V_1, V_2, \dots, V_{n_b}\}$ and attribute subsets $\mathcal{R} = \{R_1, R_2, \dots, R_{n_b}\}$. Then, PAPMI parallelizes the matrix multiplications for computing $\mathbf{P}_f^{(t)}$ and $\mathbf{P}_b^{(t)}$ from Line 2 to 9, using n_b threads in t iterations. Specifically, the i -th thread initializes $\mathbf{P}_{f_i}^{(0)}$ by $\mathbf{R}_r[:, R_i]$ and $\mathbf{P}_{b_i}^{(0)}$ by

Algorithm 15: PAPMI

Input: $\mathbf{P}, \mathbf{R}, \alpha, t, \mathcal{V}, \mathcal{R}$
Output: $\mathbf{F}', \mathbf{B}'$

- 1 Compute $\mathbf{R}_r$ and $\mathbf{R}_c$ by Equation (6.1);
- 2 **parallel for** $R_i \in \mathcal{R}$ **do**
- 3 **for** $\ell \leftarrow 1$ **to** t **do**
- 4 $\mathbf{P}_{f_i}^{(0)} \leftarrow \mathbf{R}_r[:, R_i];$
- 5 $\mathbf{P}_{b_i}^{(0)} \leftarrow \mathbf{R}_c[:, R_i];$
- 6 $\mathbf{P}_{f_i}^{(\ell)} \leftarrow (1 - \alpha) \cdot \mathbf{P} \mathbf{P}_{f_i}^{(\ell-1)} + \alpha \cdot \mathbf{P}_{f_i}^{(0)};$
- 7 $\mathbf{P}_{b_i}^{(\ell)} \leftarrow (1 - \alpha) \cdot \mathbf{P}^\top \mathbf{P}_{b_i}^{(\ell-1)} + \alpha \cdot \mathbf{P}_{b_i}^{(0)};$
- 8 Normalize $\mathbf{P}_{f_i}^{(t)}$ by columns to get $\widehat{\mathbf{P}}_{f_i}^{(t)};$
- 9 $\mathbf{h}_i \leftarrow \sum_{r_j \in R_i} \mathbf{P}_{b_i}^{(t)}[:, r_j];$
- 10 $\widehat{\mathbf{P}}_f^{(t)} \leftarrow [\mathbf{P}_{f_1}^{(t)} \cdots \mathbf{P}_{f_{n_b}}^{(t)}];$
- 11 $\mathbf{h} = \sum_{R_i \in \mathcal{R}} \mathbf{h}_i;$
- 12 $\widehat{\mathbf{P}}_b^{(t)} \leftarrow [\mathbf{P}_{b_1}^{(t)} / \mathbf{h} \cdots \mathbf{P}_{b_{n_b}}^{(t)} / \mathbf{h}];$
- 13 Initialize empty matrices $\mathbf{F}', \mathbf{B}' \in \mathbb{R}^{n \times d};$
- 14 **parallel for** $V_i \in \mathcal{V}$ **do**
- 15 $\mathbf{F}'[V_i] \leftarrow \log(n \cdot \widehat{\mathbf{P}}_f^{(t)}[V_i] + 1);$
- 16 $\mathbf{B}'[V_i] \leftarrow \log(d \cdot \widehat{\mathbf{P}}_b^{(t)}[V_i] + 1);$
- 17 **return** $\mathbf{F}', \mathbf{B}'$

$\mathbf{R}_c[:, R_i]$ (Lines 4-5), and then computes $\mathbf{P}_{f_i}^{(\ell)} = (1 - \alpha) \cdot \mathbf{P} \mathbf{P}_{f_i}^{(\ell-1)} + \alpha \cdot \mathbf{P}_{f_i}^{(0)}$ and $\mathbf{P}_{b_i}^{(\ell)} = (1 - \alpha) \cdot \mathbf{P}^\top \mathbf{P}_{b_i}^{(\ell-1)} + \alpha \cdot \mathbf{P}_{b_i}^{(0)}$ (Lines 6-7). Each thread finishes the computation by normalizing $\mathbf{P}_{f_i}^{(t)}$ by columns, *i.e.*, $\widehat{\mathbf{P}}_{f_i}^{(t)}[v_l, r_j] = \frac{\mathbf{P}_{f_i}^{(t)}[v_l, r_j]}{\sum_{v_h \in V} \mathbf{P}_{f_i}^{(t)}[v_h, r_j]}$, and summing up the columns of $\mathbf{P}_{b_i}^{(t)}$ to obtain a length- n vector $\mathbf{h}_i$, in which the v_l -th element is $\mathbf{h}_i[v_l] = \sum_{r_j \in R_i} \mathbf{P}_{b_i}^{(t)}[v_l, r_j]$ (Lines 8-9). Then, we use a main thread to aggregate the partial results of all threads at Lines 10-12. Specifically, n_b matrix blocks $\mathbf{P}_{f_i}^{(t)}$ are concatenated horizontally together as $\widehat{\mathbf{P}}_f^{(t)}$ at Line 11. After that, we sum up n_b length- n vectors $\mathbf{h}_i$ to $\mathbf{h}$, in which $\mathbf{h}[v_l] = \sum_{R_i \in \mathcal{R}} \sum_{r_j \in R_i} \mathbf{P}_{b_i}^{(t)}[v_l, r_j]$ at Line 11, and concatenate n_b matrix blocks, $\mathbf{P}_{b_i}^{(t)}[:, r_j]$ divided by $\mathbf{h}[r_j]$, horizontally together as $\widehat{\mathbf{P}}_b^{(t)}$, in which $\widehat{\mathbf{P}}_b^{(t)}[v_l, r_j] = \mathbf{P}_{b_i}^{(t)}[v_l, r_j] / \mathbf{h}[r_j]$, at Line 12. From Lines 13 to 16, PAPMI starts n_b threads to compute $\mathbf{F}'$ and $\mathbf{B}'$ block by block in parallel, based on the definitions of forward and backward node-attribute PageRank. Specifically, the i -th thread computes $\mathbf{F}'[V_i] = \log(n \cdot \widehat{\mathbf{P}}_f^{(t)}[V_i] + 1)$ and $\mathbf{B}'[V_i] = \log(d \cdot \widehat{\mathbf{P}}_b^{(t)}[V_i] + 1)$. Finally, PAPMI returns $\mathbf{F}'$ and $\mathbf{B}'$ as the approximate

forward and backward node-attribute PageRank matrices (Line 17). Lemma 6.2 indicates the accuracy guarantee of PAPMI.

Lemma 6.2. *Given same parameters $\mathbf{P}, \mathbf{R}, \alpha$ and t as inputs to Algorithm 11 and Algorithm 15, the two algorithms return the same approximate forward and backward node-attribute PageRank matrices $\mathbf{F}', \mathbf{B}'$.*

Proof. According to Lines 5-6 in Algorithm 15, we can see that

$$\mathbf{R}_r = \begin{bmatrix} \mathbf{P}_{f_1}^{(0)} & \mathbf{P}_{f_2}^{(0)} & \cdots & \mathbf{P}_{f_{n_b}}^{(0)} \end{bmatrix},$$

where $\mathbf{P}_{f_1}^{(0)}, \dots, \mathbf{P}_{f_{n_b-1}}^{(0)} \in \mathbb{R}^{n \times \frac{d}{n_b}}$ and $\mathbf{P}_{f_{n_b}}^{(0)} \in \mathbb{R}^{n \times (d\%n_b)}$, and

$$\mathbf{R}_c = \begin{bmatrix} \mathbf{P}_{b_1}^{(0)} & \mathbf{P}_{b_2}^{(0)} & \cdots & \mathbf{P}_{b_{n_b}}^{(0)} \end{bmatrix},$$

where $\mathbf{P}_{b_1}^{(0)}, \dots, \mathbf{P}_{b_{n_b-1}}^{(0)} \in \mathbb{R}^{n \times \frac{d}{n_b}}$ and $\mathbf{P}_{b_{n_b}}^{(0)} \in \mathbb{R}^{n \times (d\%n_b)}$. After t iterations, by Lines 3-10 in Algorithm 15, we have

$$\begin{aligned} \mathbf{P}_{f_i}^{(t)} &= \alpha \sum_{\ell=0}^t (1-\alpha)^\ell \mathbf{P}^\ell \mathbf{P}_{f_i}^{(0)} \text{ and} \\ \mathbf{P}_{b_i}^{(t)} &= \alpha \sum_{\ell=0}^t (1-\alpha)^\ell \mathbf{P}^{\top \ell} \mathbf{P}_{b_i}^{(0)}. \end{aligned}$$

Thus, we can derive that

$$\begin{aligned} \mathbf{P}_f^{(t)} &= \begin{bmatrix} \mathbf{P}_{f_1}^{(t)} & \cdots & \mathbf{P}_{f_{n_b}}^{(t)} \end{bmatrix} = \alpha \sum_{\ell=0}^t (1-\alpha)^\ell \mathbf{P}^\ell \mathbf{R}_r, \\ \mathbf{P}_b^{(t)} &= \begin{bmatrix} \mathbf{P}_{b_1}^{(t)} & \cdots & \mathbf{P}_{b_{n_b}}^{(t)} \end{bmatrix} = \alpha \sum_{\ell=0}^t (1-\alpha)^\ell \mathbf{P}^\ell \mathbf{R}_c \end{aligned}$$

and

$$\mathbf{h} = \sum_{R_i \in \mathcal{R}} \sum_{r_j \in R_i} \alpha \left(\sum_{\ell=0}^t (1-\alpha)^\ell \mathbf{P}^\ell \mathbf{P}_{b_i}^{(0)} \right)[:, r_j] = \sum_{r_j \in R} \mathbf{P}_b^{(t)}[:, r_j].$$

According to Inequality (6.9) and Inequality (6.10), for every pair $(v_i, r_j) \in V \times R$, $\max\{0, \mathbf{P}_f[v_i, r_j] - \epsilon\} \leq \mathbf{P}_f^{(t)}[v_i, r_j] \leq \mathbf{P}_f[v_i, r_j]$ and $\max\{0, \mathbf{P}_b[v_i, r_j] - \epsilon\} \leq \mathbf{P}_b^{(t)}[v_i, r_j] \leq$

Algorithm 16: SMGreedyInit

Input: $\mathbf{F}', \mathbf{B}', \mathcal{V}, k, t$.
Output: $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b$.

```

1 parallel for  $V_i \in \mathcal{V}$  do
2    $\Phi, \Sigma, \mathbf{V}_i \leftarrow \text{RandSVD}(\mathbf{F}'[V_i], \frac{k}{2}, t);$ 
3    $\mathbf{U}_i \leftarrow \Phi \Sigma;$ 
4  $\mathbf{V} \leftarrow [\mathbf{V}_1 \cdots \mathbf{V}_{n_b}]^\top;$ 
5  $\Phi, \Sigma, \mathbf{Y} \leftarrow \text{RandSVD}(\mathbf{V}, \frac{k}{2}, t);$ 
6  $\mathbf{W} \leftarrow \Phi \Sigma;$ 
7 parallel for  $V_i \in \mathcal{V}$  do
8    $\mathbf{X}_f[V_i] \leftarrow \mathbf{U}_i \cdot \mathbf{W}[(i-1) \cdot \frac{k}{2} : i \cdot \frac{k}{2}];$ 
9    $\mathbf{X}_b[V_i] \leftarrow \mathbf{B}'[V_i] \cdot \mathbf{Y};$ 
10   $\mathbf{S}_f[V_i] \leftarrow \mathbf{X}_f[V_i] \cdot \mathbf{Y}^\top - \mathbf{F}'[V_i];$ 
11   $\mathbf{S}_b[V_i] \leftarrow \mathbf{B}'[V_i] - \mathbf{X}_b[V_i] \cdot \mathbf{Y}^\top;$ 
12 return  $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b;$ 
    
```

$\mathbf{P}_b[v_i, r_j]$. By Lines 11-13 in Algorithm 15, for i -th block and every pair $(v_l, r_j) \in V \times R_i$,

$$\begin{aligned}
 \hat{\mathbf{P}}_f^{(t)}[v_l, r_j] &= \hat{\mathbf{P}}_{f_i}^{(t)}[v_l, r_j] = \frac{\mathbf{P}_{f_i}^{(t)}[v_l, r_j]}{\sum_{v_h \in V} \mathbf{P}_{f_i}^{(t)}[v_h, r_j]} \\
 &= \frac{\mathbf{P}_f^{(t)}[v_l, r_j]}{\sum_{v_h \in V} \mathbf{P}_f^{(t)}[v_h, r_j]}, \\
 \hat{\mathbf{P}}_b^{(t)}[v_l, r_j] &= \frac{\mathbf{P}_{b_i}^{(t)}[v_l, r_j]}{\mathbf{h}[v_l]} = \frac{\mathbf{P}_{b_i}^{(t)}[v_l, r_j]}{\sum_{R_i \in \mathcal{R}} \sum_{r_h \in R_i} \mathbf{P}_{b_i}^{(t)}[v_l, r_h]} \\
 &= \frac{\mathbf{P}_b^{(t)}[v_l, r_j]}{\sum_{r_h \in R} \mathbf{P}_b^{(t)}[v_l, r_h]}.
 \end{aligned}$$

By Lines 14-17 in Algorithm 15, the results in in Lemma 6.2 are now at hand. $\square$

6.3.2 Parallel Joint Factorization of Node-Attribute PageRank Matrices

This section presents the parallel algorithm PSVDCCD in Algorithm 17 to further improve the efficiency of the joint node-attribute PageRank matrix factorization process. At Line 1 of the algorithm, we design a parallel initialization algorithm SMGreedyInit with a split-and-merge-based parallel SVD technique for embedding vector initialization.

Algorithm 17: PSVDCCD

Input: $\mathbf{F}', \mathbf{B}', \mathcal{V}, \mathcal{R}, k, t$.
Output: $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b$.

```

1  $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b \leftarrow \text{SMGreedyInit}(\mathbf{F}', \mathbf{B}', \mathcal{V}, k, t);$ 
2 for  $\ell \leftarrow 1$  to  $t$  do
3   parallel for  $V_h \in \mathcal{V}$  do
4     for  $v_i \in V_h$  do
5       for  $l \leftarrow 1$  to  $\frac{k}{2}$  do
6         Compute  $\mu_f(v_i, l), \mu_b(v_i, l)$  by Equation (6.16);
7          $\mathbf{X}_f[v_i, l] \leftarrow \mathbf{X}_f[v_i, l] - \mu_f(v_i, l);$ 
8          $\mathbf{X}_b[v_i, l] \leftarrow \mathbf{X}_b[v_i, l] - \mu_b(v_i, l);$ 
9         Update  $\mathbf{S}_f[v_i]$  by Equation (6.18);
10        Update  $\mathbf{S}_b[v_i]$  by Equation (6.19);
11   parallel for  $R_h \in \mathcal{R}$  do
12     for  $r_j \in R_h$  do
13       for  $l \leftarrow 1$  to  $\frac{k}{2}$  do
14         Compute  $\mu_y(r_j, l)$  by Equation (6.17);
15          $\mathbf{Y}[r_j, l] \leftarrow \mathbf{Y}[r_j, l] - \mu_y(r_j, l);$ 
16         Update  $\mathbf{S}_f[:, r_j], \mathbf{S}_b[:, r_j]$  by Equation (6.20);
17 return  $\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b;$ 
    
```

Algorithm 16 shows the pseudo-code of **SMGreedyInit**, which takes as input $\mathbf{F}'$, $\mathbf{B}'$, $\mathcal{V}$, and k . Based on $\mathcal{V}$, **SMGreedyInit** splits matrix $\mathbf{F}'$ into n_b blocks and launches n_b threads. Then, the i -th thread applies randomized SVD to block $\mathbf{F}'[V_i]$ generated by the rows of $\mathbf{F}'$ based on node set $V_i \in \mathcal{V}$ (Line 1-3). After obtaining $\mathbf{V}_1, \dots, \mathbf{V}_{n_b}$, **SMGreedyInit** merges these matrices by concatenating $\mathbf{V}_1, \dots, \mathbf{V}_{n_b}$ into $\mathbf{V} = [\mathbf{V}_1 \ \dots \ \mathbf{V}_{n_b}]^\top \in \mathbb{R}^{\frac{kn_b}{2} \times d}$, and then applies **RandSVD** over it to obtain $\mathbf{W} \in \mathbb{R}^{\frac{kn_b}{2} \times \frac{k}{2}}$ and $\mathbf{Y} \in \mathbb{R}^{d \times \frac{k}{2}}$ (Lines 4-6). At Line 7, **SMGreedyInit** creates n_b threads, and uses the i -th thread to handle node subset V_i for initializing embedding vectors $\mathbf{X}_f[V_i]$ and $\mathbf{X}_b[V_i]$ at Lines 8-9, as well as computing $\mathbf{S}_f$ and $\mathbf{S}_b$ at Lines 10-11. Specifically, the forward embedding vectors of node subset V_i are initialized as $\mathbf{X}_f[V_i] = \mathbf{U}_i \cdot \mathbf{W}[(i-1) \cdot \frac{k}{2} : i \cdot \frac{k}{2}]$ at Line 8; the backward embedding vectors of V_i are initialized as $\mathbf{X}_b[V_i] = \mathbf{B}'[V_i] \cdot \mathbf{Y}$ at Line 9; $\mathbf{S}_f[V_i]$ and $\mathbf{S}_b[V_i]$ for node subset V_i are computed as $\mathbf{S}_f[V_i] = \mathbf{X}_f[V_i] \cdot \mathbf{Y}^\top - \mathbf{F}'[V_i]$ at Line 10 and $\mathbf{S}_b[V_i] = \mathbf{X}_b[V_i] \cdot \mathbf{Y}^\top - \mathbf{B}'[V_i]$ at Line 11. Finally, **SMGreedyInit** returns initialized embedding vectors $\mathbf{Y}$, $\mathbf{X}_f$, and $\mathbf{X}_b$ as well as intermediate results $\mathbf{S}_f, \mathbf{S}_b$ at Line 12. Lemma 6.3 indicates that the initial

embedding vectors produced by **SMGreedyInit** are close to that of **GreedyInit**.

After obtaining $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$ by **SMGreedyInit**, Lines 2-16 in Algorithm 17 train embedding vectors by cyclic coordinate descent in parallel based on subsets $\mathcal{V}$ and $\mathcal{R}$, in t iterations. In each iteration, **PSVDCCD** first fixes $\mathbf{Y}$ and launches n_b threads to update $\mathbf{X}_f$ and $\mathbf{X}_b$ in parallel by blocks according to $\mathcal{V}$, and then updates $\mathbf{Y}$ using the n_b threads in parallel by blocks according to $\mathcal{R}$, with $\mathbf{X}_f$ and $\mathbf{X}_b$ fixed (Lines 11-16). Specifically, from Lines 4 to 10, each thread is assigned a node subset V_h , and for each node v_i in V_h , the thread updates $\mathbf{X}_f[v_i], \mathbf{X}_b[v_i]$ based on Equations (6.13) and (6.14), and then updates $\mathbf{S}_f[v_i]$ and $\mathbf{S}_b[v_i]$ by Equations (6.18) and (6.19). After updating $\mathbf{X}_f$ and $\mathbf{X}_b$, each thread is assigned an attribute subset R_h to update $\mathbf{Y}[r_j]$ according to Equation (6.15) and $\mathbf{S}[:, r_j]$ by Equation (6.20) for each attribute $r_j \in R_h$ (Lines 12-16). Finally, Algorithm 17 returns $\mathbf{X}_f$ and $\mathbf{X}_b$ as forward and backward embedding vectors, and $\mathbf{Y}$ as the attribute embedding vectors.

Lemma 6.3. *Given same $\mathbf{F}', \mathbf{B}', k$ and t as inputs to Algorithm 12 and Algorithm 16, the outputs $\mathbf{X}_f, \mathbf{Y}, \mathbf{S}_f, \mathbf{S}_b$ returned by both algorithms satisfy that $\mathbf{X}_f \cdot \mathbf{Y}^\top = \mathbf{F}', \mathbf{Y}^\top \mathbf{Y} = \mathbf{I}$ and $\mathbf{S}_f = \mathbf{S}_b \mathbf{Y} = \mathbf{0}$, when $t = \infty$.*

Proof. Let the outputs of Algorithm 12 be $\mathbf{X}_f, \mathbf{X}_b, \mathbf{Y}, \mathbf{S}_f$ and $\mathbf{S}_b$, and the results returned by Algorithm 16 be $\hat{\mathbf{X}}_f, \hat{\mathbf{X}}_b, \hat{\mathbf{Y}}$ and $\hat{\mathbf{S}}_f, \hat{\mathbf{S}}_b$. According to [104], $t = \infty$ implies that **RandSVD** produces the same factorized results as that returned by exact SVD. Therefore, $\mathbf{X}_f \cdot \mathbf{Y}^\top = \mathbf{F}', \mathbf{S}_f = \mathbf{0}, \mathbf{X}_b = \mathbf{B}' \mathbf{Y}$ and $\mathbf{Y}$ is unitary, *i.e.*, $\mathbf{Y}^\top \mathbf{Y} = \mathbf{I}$. This leads to $\mathbf{S}_b \mathbf{Y} = (\mathbf{X}_b \mathbf{Y}^\top - \mathbf{B}') \mathbf{Y} = \mathbf{0}$.

On the other hand, we consider Algorithm 16. Based on Lines 2-3 and Lines 5-6, we have $\mathbf{U}_i \mathbf{V}_i^\top = \mathbf{F}'[V_i]$, $\mathbf{W} \hat{\mathbf{Y}}^\top = \mathbf{V}$ and unitary matrix $\hat{\mathbf{Y}}$, *i.e.*, $\hat{\mathbf{Y}}^\top \hat{\mathbf{Y}} = \mathbf{I}$. Then by Line 8

and Line 10, we derive that

$$\begin{aligned}
 \mathbf{F}' &= \begin{bmatrix} \mathbf{F}'[V_1] \\ \mathbf{F}'[V_1] \\ \vdots \\ \mathbf{F}'[V_{n_b}] \end{bmatrix} = \begin{bmatrix} \mathbf{U}_1 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_2 & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{U}_{n_b} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{V}_1^\top \\ \mathbf{V}_2^\top \\ \vdots \\ \mathbf{V}_{n_b}^\top \end{bmatrix} \\
 &= \begin{bmatrix} \mathbf{U}_1 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_2 & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{U}_{n_b} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{W}_1 \\ \mathbf{W}_2 \\ \vdots \\ \mathbf{W}_{n_b} \end{bmatrix} \cdot \hat{\mathbf{Y}}^\top \\
 &= \begin{bmatrix} \hat{\mathbf{X}}_f[V_1] \\ \hat{\mathbf{X}}_f[V_2] \\ \vdots \\ \hat{\mathbf{X}}_f[V_{n_b}] \end{bmatrix} \cdot \hat{\mathbf{Y}}^\top = \hat{\mathbf{X}}_f \cdot \hat{\mathbf{Y}}^\top,
 \end{aligned}$$

and thus $\hat{\mathbf{S}}_f = \mathbf{0}$. In addition, according to Line 9 and Line 11, we have $\hat{\mathbf{X}}_b = \mathbf{B}'\hat{\mathbf{Y}}$ and $\hat{\mathbf{S}}_b\hat{\mathbf{Y}} = (\hat{\mathbf{X}}_b\hat{\mathbf{Y}}^\top - \mathbf{B}')\hat{\mathbf{Y}} = \mathbf{0}$. The proof is complete. $\square$

6.3.3 Complexity Analysis

Observe that the non-parallel parts of Algorithms 15 and 16 take $O(nd)$ time, as each of them performs a constant number of operations on $O(nd)$ matrix entries. Meanwhile, for the parallel parts of Algorithms 15 and 17, each thread runs in $O\left(\frac{md}{n_b} \cdot \log\left(\frac{1}{\epsilon}\right)\right)$ and $O\left(\frac{ndkt}{n_b}\right)$ time, respectively, since we divide the workload evenly to n_b threads. Specifically, each thread in Algorithm 15 runs in $O\left(\frac{md}{n_b} \cdot \log\left(\frac{1}{\epsilon}\right)\right)$ time. Algorithm 17 first takes $O\left(\frac{n}{n_b}dkt\right)$ time for each thread to factorize a $\frac{n}{n_b} \times d$ matrix block of $\mathbf{F}'$ (Lines 1-3 in Algorithm 16). In addition, Lines 4-6 in Algorithm 16 requires $O(n_bdk)$ time. In merge course (*i.e.*, Lines 7-11 in Algorithm 16), the matrix multiplications take $O\left(\frac{n}{n_b}k^2\right)$ time. In the t iterations of CCD (*i.e.*, Lines 2-16 in Algorithm 17), each thread spends $O\left(\frac{ndkt}{n_b}\right)$ time to update. Thus, the computational time complexity per thread in Algorithm 14 is

$$O\left(\frac{md + ndk}{n_b} \cdot \log\left(\frac{1}{\epsilon}\right)\right).$$

Algorithm 15 and Algorithm 17 require $O(m + nd)$ and $O(nd)$ space, respectively. Therefore, the space complexity of PANE is $O(m + nd)$.

Table 6.3: Datasets. ($K=10^3$, $M=10^6$)

Name	$ V $	$ E_V $	$ R $	$ E_R $	$ L $	Refs
<i>Cora</i>	2.7K	5.4K	1.4K	49.2K	7	[97, 101, 109, 161, 164, 187]
<i>Citeseer</i>	3.3K	4.7K	3.7K	105.2K	6	[97, 101, 109, 161, 164, 187]
<i>Facebook</i>	4K	88.2K	1.3K	33.3K	193	[94, 101, 167, 181]
<i>Pubmed</i>	19.7K	44.3K	0.5K	988K	3	[101, 109, 181, 187]
<i>Flickr</i>	7.6K	479.5K	12.1K	182.5K	9	[101]
<i>Google+</i>	107.6K	13.7M	15.9K	300.6M	468	[94, 167]
<i>TWeibo</i>	2.3M	50.7M	1.7K	16.8M	8	-
<i>MAG</i>	59.3M	978.2M	2K	434.4M	100	-

6.4 Experiments

Table 6.4: Attribute inference performance.

Method	<i>Cora</i>	<i>Citeseer</i>	<i>Facebook</i>	<i>Pubmed</i>	<i>Flickr</i>	<i>Google+</i>	<i>TWeibo</i>	<i>MAG</i>
	AUC	AUC	AUC	AUC	AUC	AUC	AUC	AUC
BLA	0.559	0.540	0.653	0.520	0.660	-	-	-
CAN	0.865	0.875	0.765	0.734	0.772	-	-	-
PANE (single thread)	0.913	0.903	0.828	0.871	0.825	0.972	0.774	0.876
PANE (parallel)	0.909	0.899	0.825	0.867	0.822	0.969	0.773	0.874

We experimentally evaluate our proposed method PANE (both single-thread and parallel versions) against 10 competitors on three tasks: link prediction, attribute inference and node classification, using 8 real datasets. All experiments are conducted on a Linux machine powered by an Intel Xeon(R) E7-8880 v4@2.20GHz CPUs and 1TB RAM. All algorithms are collected from their respective authors and are implemented in Python, except NRP, TADW and LQANR. To achieve a fair comparison in efficiency evaluation, we implement TADW and LQANR in Python based on the Matlab implementations provided by their respective authors.

6.4.1 Experiments Setup

Datasets. Table 6.3 lists the statistics of the datasets used in our experiments. All graphs are directed except *Facebook* and *Flickr*. $|V|$ and $|E_V|$ denote the number of nodes and edges in the graph, whereas $|R|$ and $|E_R|$ represent the number of attributes

Table 6.5: Link prediction performance.

Method	<i>Cora</i>	<i>Citeseer</i>	<i>Pubmed</i>	<i>Facebook</i>	<i>Flickr</i>	<i>Google+</i>	<i>TWeibo</i>	<i>MAG</i>
	AUC	AUC	AUC	AUC	AUC	AUC	AUC	AUC
NRP	0.796	0.86	0.87	0.969	0.909	0.989	0.967	0.915
GATNE	0.791	0.687	0.745	0.961	0.805	-	-	-
TADW	0.829	0.895	0.904	0.752	0.573	-	-	-
ARGA	0.64	0.637	0.623	0.71	0.676	-	-	-
BANE	0.875	0.899	0.919	0.796	0.64	0.56	-	-
PRRE	0.879	0.895	0.887	0.899	0.789	-	-	-
STNE	0.808	0.71	0.789	0.962	0.638	-	-	-
CAN	0.663	0.734	0.734	0.714	0.5	-	-	-
DGI	0.51	0.5	0.73	0.711	0.769	0.792	0.721	-
LQANR	0.886	0.916	0.904	0.951	0.824	-	-	-
PANE (single thread)	0.933	0.932	0.985	0.982	0.929	0.987	0.976	0.96
PANE (parallel)	0.929	0.929	0.985	0.98	0.927	0.984	0.975	0.958

and the number of node-attribute associations (*i.e.*, the number of nonzero entries in attribute matrix $\mathbf{R}$). In addition, L is the set of *node labels*, which are used in the node classification task. *Cora*³, *Citeseer*³, *Pubmed*³ and *Flickr*⁴ are benchmark datasets used in prior work [67, 97, 101, 109, 161, 187]. *Facebook*⁵ and *Google+*⁵ are social networks used in [94]. For *Facebook* and *Google+*, we treat each ego-network as a label and extract attributes from their user profiles, which is consistent with the experiments in prior work [101, 167]

To evaluate the scalability of the proposed solution, we also introduce two new datasets *TWeibo*⁶ and *MAG*⁷ that have not been used in previous ANE papers due to their massive sizes. *TWeibo* [83] is a social network, in which each node represents a user, and each directed edge represents a following relationship. We extract the 1657 most popular tags and keywords from its user profile data as the node attributes. The labels are generated and categorized into eight types according to the ages of users. *MAG* dataset is extracted from the well-known *Microsoft Academic Knowledge Graph* [126], where each node represents a paper and each directed edge represents a citation. We extract 2000 most frequently used distinct words from the abstract of all papers as the attribute set and regard the fields of study of each paper as its labels.

³<http://linqs.soe.ucsc.edu/data>

⁴<https://github.com/mengzaiqiao/CAN>

⁵<http://snap.stanford.edu/data>

⁶<https://www.kaggle.com/c/kddcup2012-track1>

⁷<http://ma-graph.org/rdf-dumps/>

Figure 6.2: Node classification results (best viewed in color).

Baselines and Parameter Settings. We compare our methods PANE (single thread) and PANE (parallel) against 10 state-of-the-art competitors: eight recent ANE methods including BANE [164], CAN [101], STNE [97], PRRE [187], TADW [161], ARGA [109], DGI [139] and LQANR [163], one state-of-the-art homogeneous network embedding method NRP [168], and one latest attributed heterogeneous network embedding algorithm GATNE [15]. All methods except PANE (parallel) run on a single CPU core. Note that although GATNE itself is a parallel algorithm, its parallel version requires the proprietary AliGraph platform which is not available to us.

The parameters of all competitors are set as suggested in their respective papers. For PANE (single thread) and PANE (parallel), by default we set error threshold $\epsilon = 0.015$ and random walk stopping probability $\alpha = 0.5$, and we use $n_b = 10$ threads for PANE (parallel). Unless otherwise specified, we set space budget $k = 128$.

The evaluation results of our proposed methods against the competitors for attribute inference, link prediction and node classification, are reported in Sections 6.4.2, 6.4.3 and 6.4.4 respectively. The efficiency and scalability evaluation is reported in Section 6.4.5. A method will be excluded if it cannot finish training within one week. Section 6.4.6 analyzes the parameter sensitivities of PANE. Section 6.4.7 studies the effectiveness of GreedyInit in PANE.

6.4.2 Attribute Inference

Attribute inference aims to predict the values of attributes of nodes. Note that, except for CAN [101], none of the other competitors is capable of performing attribute inference, since they only generate embedding vectors for nodes, not attributes. Hence, we compare our solutions against CAN for attribute inference. Further, we compare against BLA, the state-of-the-art attribute inference algorithm [160]. Note that BLA is not an ANE solution.

We split the nonzero entries in the attribute matrix $\mathbf{R}$, and regard 20% as the test set $\mathbf{R}_{test}$ and the remaining 80% part as the training set $\mathbf{R}_{train}$. CAN runs over $\mathbf{R}_{train}$ to generate node embedding vector $\mathbf{X}[v_i]$ for each node $v_i \in V$ and attribute embedding vector $\mathbf{Y}[r_j]$ for each attribute $r_j \in R$. Following [101], we use the inner product of $\mathbf{X}[v_i]$ and $\mathbf{Y}[r_j]$ as the predicted score of attribute r_j with respect to node v_i . Note that PANE generates a forward embedding vector $\mathbf{X}_f[v_i]$ and a backward embedding vector $\mathbf{X}_b[v_i]$ for each node $v_i \in V$, and also an attribute embedding vector $\mathbf{Y}[r_j]$ for each attribute $r_j \in R$. Based on objective function in Equation (6.4), $\mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_j]^\top$ is expected to preserve forward node-attribute PageRank value $\mathbf{F}[v_i, r_j]$, and $\mathbf{X}_b[v_i] \cdot \mathbf{Y}[r_j]^\top$ is expected to preserve backward node-attribute PageRank value $\mathbf{B}[v_i, r_j]$. Thus, we predict the score between v_i and r_j through the node-attribute PageRank between node v_i and attribute r_j , including both forward node-attribute PageRank and backward node-attribute PageRank, denoted as $p(v_i, r_j)$, by utilizing their embedding vectors as follows.

$$p(v_i, r_j) = \mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_j]^\top + \mathbf{X}_b[v_i] \cdot \mathbf{Y}[r_j]^\top \approx \mathbf{F}[v_i, r_j] + \mathbf{B}[v_i, r_j]. \quad (6.21)$$

Following prior work [101], we adopt the *Area Under Curve* (AUC) metric to measure the performance.

Table 6.4 presents the attribute inference performance of PANE (single thread), PANE (parallel), CAN and BLA. Observe that PANE (single thread) consistently achieves the best performance on all datasets and significantly outperforms existing solutions by a large margin, demonstrating the power of forward node-attribute PageRank and backward node-attribute PageRank that are preserved in embedding vectors $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$, to capture the node-attribute PageRank between nodes and attributes in attributed networks. For instance, on *Pubmed*, PANE (single thread) has high accuracy 0.871 AUC,

while that of CAN is only 0.734 respectively. Further, CAN and BLA fail to process large attributed networks *Google+*, *TWeibo* and *MAG* in one week, and, thus are not reported. Observe that parallel PANE has close performance (*i.e.*, AUC) to that of PANE (single thread). For instance, on *Pubmed*, the difference of AUC between PANE (single thread) and PANE (parallel) is just 0.004. This negligible difference is introduced by the split-merge-based parallel SVD technique **SMGreedyInit** for matrix decomposition. As shown in Section 6.4.5, parallel PANE is considerably faster than PANE (single thread) by up to 9 times, while obtaining almost the same accuracy performance.

6.4.3 Link Prediction

Link prediction aims to predict the edges that are most likely to form between nodes. We first randomly remove 30% edges in input graph G , obtaining a residual graph G' and a set of the removed edges. We then randomly sample the same amount of non-existing edges as negative edges. The test set E' contains both the removed edges and the negative edges. We run PANE and all competitors on the residual graph G' to produce embedding vectors, and then evaluate the link prediction performance with E' as follows. PANE produces the a forward embedding vector $\mathbf{X}_f[v_i]$ and a backward embedding vector $\mathbf{X}_b[v_i]$ for each node $v_i \in V$, as well as an attribute embedding vector $\mathbf{Y}[r_l]$ for each attribute $r_l \in R$. As explained, $\mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_l]^\top$ preserves $\mathbf{F}[v_i, r_l]$, and $\mathbf{X}_b[v_j] \cdot \mathbf{Y}[r_l]^\top$ preserves $\mathbf{B}[v_j, r_l]$. Recall that $\mathbf{F}[v_i, r_l]$ measures the affinity from v_i to r_l over the attributed network; similarly given node v_j and attribute r_l , $\mathbf{B}[v_j, r_l]$ measures the affinity from r_l to v_j over the network. Intuitively, $\mathbf{F}[v_i, r_l] \cdot \mathbf{B}[v_j, r_l]$ represents the affinity from node v_i to node v_j based on attribute r_l . The affinity between nodes v_i and v_j , denoted as $p(v_i, v_j)$, can be evaluated by summing up the affinity between the two nodes over all attributes in R , which can be computed as follows and indicates the possibility of forming an edge from v_i to v_j .

$$p(v_i, v_j) = \sum_{r_l \in R} (\mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_l]^\top) \cdot (\mathbf{X}_b[v_j] \cdot \mathbf{Y}[r_l]^\top) \approx \sum_{r_l \in R} \mathbf{F}[v_i, r_l] \cdot \mathbf{B}[v_j, r_l]. \quad (6.22)$$

Therefore, for PANE, we can calculate $p(v_i, v_j)$ as the prediction score of the directed edge (v_i, v_j) . NRP generates a forward embedding vector $\mathbf{X}_f[v_i]$ and a backward embedding

vector $\mathbf{X}_b[v_i]$ for each node v_i and uses $p(v_i, v_j) = \mathbf{X}_f[v_i] \cdot \mathbf{X}_b[v_j]^\top$ as the prediction score for the directed edge (v_i, v_j) [168]. For undirected graphs, PANE (single thread), PANE (parallel) and NRP utilize $p(v_i, v_j) + p(v_j, v_i)$ as the prediction score for the undirected edge between v_i and v_j . In terms of the remaining competitors that only work for undirected graphs, they learn one embedding vector $\mathbf{X}[v_i]$ for each node v_i . In literature, there are four ways to calculate the link prediction score $p(v_i, v_j)$, including *inner product* method used in CAN and ARG, *cosine similarity* method used in PRRE and ANRL, *Hamming distance* method used in BANE, as well as *edge feature* method used in [60, 99]. We adopt all these four prediction methods over each competitor and report the competitor’s best performance on each dataset. Following previous work [101, 109], we use *Area Under Curve* (AUC) to evaluate link prediction accuracy.

Table 6.5 reports the AUC and AP scores of each method on each dataset. PANE (single thread) consistently outperforms all competitors over all datasets except NRP on *Google+*, by a substantial margin of up to 6.6% for AUC. For large attributed networks including *Google+*, *TWeibo* and *MAG*, most existing solutions fail to finish processing within a week and thus are not reported. The superiority of PANE (single thread) over competitors is achieved by (i) learning a forward embedding vector and a backward embedding vector for each node to capture the asymmetric transitivity (*i.e.*, edge direction) in directed graphs, and (ii) combining both node embedding vectors and attribute embedding vectors together for link prediction in Equation (6.22), with the consideration of both topological and attribute features. On *Google+*, NRP is slightly better than PANE (single thread), since *Google+* has more than 15 thousand attributes (see Table 6.3), leading to some accuracy loss when factorizing forward and backward node-attribute PageRank matrices into low dimensionality $k = 128$ by PANE (single thread). As shown in Table 6.5, our parallel PANE also outperforms all competitors significantly except NRP on *Google+*, and parallel PANE has comparable performance with PANE (single thread) over all datasets. As reported later in Section 6.4.5, parallel PANE is significantly faster than PANE (single thread) by up to 9 times, with almost the same accuracy performance for link prediction.

Figure 6.3: Running time (best viewed in color).

6.4.4 Node Classification

Node classification predicts the node labels. Note that *Facebook*, *Google+* and *MAG* are multi-labelled, meaning that each node can have multiple labels. We first run PANE (single thread), PANE and the competitors on the input attributed network G to obtain their embedding vectors. Then we randomly sample a certain number of nodes (ranging from 10% to 90%) to train a linear support-vector machine (SVM) classifier [27] and use the rest for testing. NRP, PANE (single thread), and PANE generate a forward embedding vector $\mathbf{X}_f[v_i]$ and a backward embedding vector $\mathbf{X}_b[v_i]$ for each node $v_i \in V$. So we normalize the forward and backward embedding vectors of each node v_i , and then concatenate them as the feature representation of v_i to be fed into the classifier. Akin to prior work [73, 101, 163], we use Micro-F1 and Macro-F1 to measure node classification performance. We repeat for 5 times and report the average performance.

Figure 6.2 shows the Micro-F1 results when varying the percentage of nodes used for training from 10% to 90% (*i.e.*, 0.1 to 0.9). The results of Macro-F1 are similar and thus omitted for brevity. Both versions of PANE consistently outperform all competitors on all datasets, which demonstrates that our proposed solutions effectively capture the topology and attribute information of the input attributed networks. Specifically, compared with the competitors, PANE (single thread) achieves a remarkable improvement, up to 3.7% on *Cora*, *Citeseer*, *Pubmed* and *Flickr*, and up to 11.6% on *Facebook*. On the large attributed networks *Google+*, *TWeibo* and *MAG*, most existing solutions fail to finish within a week and thus their results are omitted. Furthermore, PANE (single thread) outperforms NRP

Figure 6.4: Running time with varying parameters.

by at least 3.4% and 6% on *Google+* and *TWeibo* as displayed in Figures 6.2(f) and 6.2(g), respectively. In addition, PANE (single thread) and PANE (parallel) are superior to NRP with a significant gain up to 17.2% on *MAG*. Over all datasets, PANE (parallel) has similar performance to that of PANE (single thread), while as shown in Section 6.4.5, PANE (parallel) is significantly faster than PANE (single thread).

6.4.5 Efficiency and Scalability

Figure 6.3(a) and Figure 6.3(b) together report the running time required by each method on all datasets. The y -axis is the running time (seconds) in log-scale. The reported running time does not include the time for loading datasets and outputting embedding vectors. We omit any methods with processing time exceeding one week. For completeness, We report the running time of NRP, but note that NRP is designed only for homogeneous networks and not capable of handling attributed networks.

Both versions of PANE are significantly faster than all ANE competitors, often by orders of magnitude. For instance, on *Pubmed* in Figure 6.3(a), PANE takes 1.1 seconds and PANE (single thread) requires 8.2 seconds, while the fastest ANE competitor TADW consumes 405.3 seconds, meaning that PANE (single thread) (resp. PANE) is $49\times$ (resp. $368\times$) faster. On large attributed networks including *Google+*, *TWeibo*, and *MAG*, most existing ANE solutions cannot finish within a week, while our proposed solutions PANE (single thread) and PANE are able to handle such large-scale networks efficiently. PANE is up to 9 times faster than PANE (single thread) over all datasets. For instance, on *MAG* dataset that has 59.3 million nodes, when using 10 threads, PANE requires 11.9 hours while PANE (single thread) costs about five days, which indicates the power of our parallel techniques in Section 6.3.

Figure 6.5: Attribute inference results with varying parameters (best viewed in color).

Figure 6.6: Link prediction results with varying parameters (best viewed in color).

Figure 6.4(a) plots the speedups of parallel PANE over single-thread PANE on *Google+* and *TWeibo* when varying the number of threads n_b from 1 to 20, while Figure 6.4(b) and Figure 6.4(c) plot the running time of PANE when varying space budget k from 16 to 256 and error threshold ϵ from 0.001 to 0.25, respectively. When one of these parameters varies, we keep others as default settings described in Section 6.4.1. In Figure 6.4(a), when n_b increases, the speedups of parallel PANE over PANE improves significantly, demonstrating the runtime linearity of PANE with respect to the number of threads n_b . When k is increased from 16 to 256 in Figure 6.4(b), the running time is quite stable and goes up slowly, showing the efficiency robustness of our solution. In Figure 6.4(c), the running time of PANE decreases considerably when increasing ϵ in $\{0.001, 0.005, 0.015, 0.05, 0.25\}$. When ϵ increases from 0.001 to 0.25, the running time on *Google+* and *TWeibo* reduces by about 10 times, which is consistent with our analysis that PANE runs in linear to $\log(1/\epsilon)$ in Section 6.3.

6.4.6 Paramter Analysis

PANE involves several parameters including embedding dimensionality k , the number of threads n_b , error threshold ϵ and random walk stopping probability α . We study the effects of varying these parameters for attribute inference and link prediction on *Cora*, *Citeseer*, *Facebook*, *Pubmed* and *Flickr*, respectively. We report AUC results in Figure 6.5 and Figure 6.6 when one of the parameters is varied, the others are kept as default values in Section 6.4.1.

Figure 6.5(a) and Figure 6.6(a) display the AUC scores of PANE with regard to attribute inference and link prediction on five graphs respectively when varying space budget k in $\{16, 32, 64, 128, 256\}$. It can be observed that the AUC grows notably when k increases from 16 to 256. This indicates that a large space budget benefits to producing more accurate embedding vectors. Figure 6.5(b) and Figure 6.6(b) depict the AUC scores when varying the number of threads n_b from 1 to 20. We observe that the attribute inference performance and the link prediction performance decreases slowly as n_b increases. This is due to that PANE performs SVD over each matrix block and each factorization introduces an error, and thus the larger n_b is, the less accurate the embedding vectors are. Note that when $\alpha = 0.5$, varying ϵ from 0.001 to 0.25 corresponds to reducing the number of iterations t from 9 to 1. From Figure 6.5(c) and Figure 6.6(c), we can see that the attribute inference and link prediction performance are nearly stationary when increasing ϵ from 0.001 to 0.05, while the performance declines rapidly when ϵ is beyond 0.05. From Figure 6.5(d) and 6.6(d), where random walk stopping probability α is varied from 0.1 to 0.9, we note that PANE achieves the best performance in terms of link prediction and attribute inference on *Cora*, *Facebook* and *Pubmed* when $\alpha = 0.5$ and that on *Citeseer* and *Flickr* when $\alpha = 0.7$. The performance first increases and then downgrades as α is increased owing to that if α is too small, PANE tends to detect distant nodes; at the other extreme if α is too large, only limited local neighborhoods of nodes are preserved in the embedding vectors. As a result, picking $\alpha = 0.5$ yields favorable performance.

6.4.7 Effectiveness Evaluation of GreedyInit

In this set of experiments, we evaluate the effectiveness of GreedyInit, by comparing it with a random initialization, as shown in Figures 6.7 and 6.8. First, let PANE-R be

Figure 6.7: Effectiveness of GreedyInit in Link Prediction.

Figure 6.8: Effectiveness of GreedyInit in Attribute Inference.

the algorithm that uses a random initialization, replacing GreedyInit of PANE (Line 1 in Algorithm 13). Figures 6.7 and 6.8 plot the running time (x -axis) *v.s.* AUC (y -axis) for link prediction and attribute inference tasks respectively, on *Facebook*, *Pubmed* and *Flickr* datasets, when varying the number of iterations t for the cyclic coordinate descent (Line 2 in Algorithm 4 of the revised paper) in $\{1, 2, 5, 10, 20\}$. From Figure 6.7 and 6.8, we observe that, as the number of iterations t increases, both PANE and PANE-R require more running time to finish, and also achieve higher link prediction and attribute inference performance. This is due to that using more iterations, we are able to obtain high-quality embeddings that are closer to the optimal solution of the objective function in Equation (6.4). Particularly, our main observation from Figure 6.7 and 6.8 is that PANE consistently outperforms PANE-R over *Facebook*, *Pubmed* and *Flickr*. Given the same amount of time, PANE-R always has a lower AUC than PANE. In other words, to achieve the same AUC as PANE, PANE-R takes a longer time (more iterations). For instance, in Figure 6.8(b), PANE achieves 0.87 AUC score, using only 5 seconds, while PANE-R requires 12 seconds. As a result, we can draw the conclusion that GreedyInit is effective to converge quickly and produce high-quality results, which is consistent with

our claim in Section 6.2.2.

6.5 Summary

This chapter presents **PANE**, an effective solution for ANE computation that scales to massive graphs with tens of millions of nodes, while obtaining state-of-the-art result utility. The high scalability and effectiveness of **PANE** are mainly due to a novel problem formulation based on a random walk model, a highly efficient and sophisticated solver, and non-trivial parallelization. Extensive experiments show that **PANE** achieves substantial performance enhancements over state-of-the-arts in terms of both efficiency and result utility.

Chapter 7

Conclusion and Future Work

7.1 Conclusions

In this thesis, we study how to exploit PageRank-based techniques to address three problems over large-scale graphs efficiently and effectively, *i.e.*, local graph clustering, homogeneous network embedding and attributed network embedding, all of which find enormous real-world applications.

First, in Chapter 4, we propose two novel heat-kernel-based local clustering algorithms, TEA and TEA+, for computing approximate heat kernel PageRank values and local graph clustering efficiently. Our algorithms bridge deterministic graph traversal with Monte-Carlo random walks in a non-trivial way, thereby overcoming the drawbacks of both and achieving significant gain in performance in comparison to the state-of-the-art local clustering techniques. Our experiments demonstrate that TEA+ significantly outperforms the state-of-the-art heat-kernel-based algorithm by at least 4 times on small graphs and up to one order of magnitude on large graphs in terms of computational time when producing clusters with the same qualities.

Next, in Chapter 5, we present NRP, a novel, efficient and effective approach for homogeneous network embedding. NRP constructs embedding vectors based on personalized PageRank and reweights each node's embedding vectors based on an objective function concerning the in-/out- degree of each node. We show that NRP runs in time almost linear to the size of the input graph, and that it requires less than four hours to process a graph with 1.2 billion edges. Extensive experiments on real data also demonstrate that

NRP considerably outperforms the state of the arts in terms of the accuracy on node classification, link prediction, and graph reconstruction tasks.

In Chapter 6, we presents PANE, an effective solution for attributed network embedding computation that scales to massive graphs with tens of millions of nodes, while obtaining state-of-the-art result utility. The high scalability and effectiveness of PANE are mainly due to a novel problem formulation based on a node-attribute PageRank model, a highly efficient and sophisticated solver, and non-trivial parallelization. Extensive experiments show that PANE achieves substantial performance enhancements over state-of-the-arts in terms of both efficiency and result utility.

7.2 Future Directions

In this section, we discuss three directions that are related to our three studied problems for future work. The first direction is to extend our heat-kernel-based local graph clustering solution to handle attributed graphs. In many scenarios, nodes in the graph might be associated with rich attributes, for example, genders, ages and locations of users in social networks, which can be very useful in community detection. However, in our first problem, *i.e.*, local graph clustering, only the topology information of the graph, *i.e.*, heat kernel PageRank, is considered in the generation of clusters, which may produce suboptimal results. In terms of network embedding, we studied homogeneous network embedding and attributed network embedding. We show how to compute PageRank-like values between nodes or between nodes and attributes efficiently, and then based on which learn high-quality embedding vectors for nodes or attributes. Nevertheless, there are numerous scenarios with nodes and/or edges of various types. For example, in Taobao, a node can be a customer, a product or a seller, as well as an edge may represent a rating relationship, or buying and selling relationships. Embedding such attributed heterogeneous networks is difficult because of the bewildering combination of heterogeneous contents and structures, which is our second direction of study. Furthermore, extending our solutions for processing dynamic graphs efficiently is our third direction. In our proposed solutions, graphs are assumed to be static, which is often not true in practice. For example, the Twitter social network is highly dynamic with 9% of

all edges (*i.e.*, 153 million edges) changing every month [105], namely, almost 60 updates on edges every second on average. As a result, it would incur prohibitive overheads if we directly apply our proposed solutions, which results in terrible experiences in online services. Towards this end, we propose to devise efficient and accurate algorithms for updating reweighted personalized PageRank and node-attributed PageRank values for our proposed embedding solutions, thereby handling dynamic graphs efficiently, especially sizable ones.

7.2.1 Local Clustering on Attributed Graphs.

Recall that the metric for measuring the clustering quality in our first problem is conductance, which is defined as the ratio of the number of the cluster's edge cuts to the number of the cluster's total edges, and is only determined by the topology of the cluster and graph. However, in attributed graphs, nodes with similar attributes may also be likely to cluster together, although they are usually not directly connected but reside in distant positions. Recall that our proposed methods, *i.e.*, TEA and TEA+, first compute heat kernel PageRank scores of all other nodes from the perspective of the given source node and then find the cluster with minimal conductance by performing a sweep over the nodes sorted by the degree-normalized heat kernel PageRank scores in descending order. The heat kernel PageRank scores are estimated by a deterministic graph traversal followed by some random walks, which are fully determined by the neighborhood of the nodes regardless of their attributes. To overcome the limitations of TEA and TEA+, naive thinking is to devise a technique in which random walks over the graph are guided by both attribute and structure information, meaning that we need to pre-compute the attribute similarity of any given node pair and decide the next jump in a random walk by choosing one of the most attribute-similar nodes or a direct neighbor in a certain way. Note that we do not know which nodes are most similar to a given node in advance, implying that we need to compute the similarities of all nodes with regard to a given node and thereby leading to a substantial cost for all nodes, *i.e.*, $O(n^2)$ time in total. Consequently, reducing the cost of computing the attribute similarities is the key to improve the efficiency of our new algorithm. To address this issue, we can just ignore distant nodes in the graph since they have negligible heat kernel PageRank scores and

may never be considered during the course of random walks. We will try to find out specific solutions for random walks and the computation of attribute similarities in the future.

7.2.2 Attributed Heterogeneous Network Embedding

Existing methods mainly focus on graphs with single-typed nodes/edges and fail to handle large-scale graphs. Numerous real-world graphs are heterogeneous, which consist of billions of nodes and edges of multiple types, and each node is associated with various attributes. In [16], Chang *et al.* propose to jointly take the contents and topological structures in networks into account and represent different objects in heterogeneous networks to unified vector representations. Recently, Alibaba released an industrial distributed computation system for graph neural networks, *i.e.*, AliGraph [189], built on which, GATNE [15] is proposed to tackle attributed heterogeneous network embedding problem. For each node and edge type, GATNE learns node embeddings with Skip-Gram model by sampling biased random walks. Finally, it obtains overall node embeddings by concatenating that for all types. However, it incurs expensive training overheads and highly relies on the power of the distributed system. As shown in Chapter 5 and Chapter 6, PageRank-based embedding techniques, NRP and PANE, are efficient and effective to capture the node-node affinity or node-attribute affinity over the graphs. However, both of them are designed for graphs with single-typed nodes/edges. Simply treating all nodes/edges in the heterogeneous graph as the same type and applying PANE to it leads to severely compromised result utility. Moreover, classic solutions for heterogeneous network embedding [33, 76] pre-defined some meta-path rules to guide the random walks over the graphs, which highly rely on expert knowledge and are limited in capturing complex semantics on multi-typed nodes and edges. To embed attributed heterogeneous networks effectively and efficiently, the basic idea of our sketched approach is to extend the basic node-attributed random walk model in PANE, which only considers the relationship between a node and an attribute from a single perspective, to a multi-dimensional random walk model, namely each node/edge type is considered as a dimension and multi-dimensional relationships between multi-typed nodes and attributes are considered. By doing so, the objective function in PANE can be transformed into

a tensor-factorization-based problem. It would be highly challenging and interesting to devise techniques to handle this tensor-factorization-based problem efficiently.

7.2.3 Dynamic Homogeneous/Attributed Network Embedding

Although our proposed embedding solutions for static homogeneous graphs and static attributed graphs, *i.e.*, **NRP** and **PANE**, are very efficient, they still require expensive overheads when processing a highly dynamic large graph. More specifically, recall that both **NRP** and **PANE** involve approximating PageRank-based proximity matrices and generating initial embedding via singular value decomposition (SVD). As aforementioned, graph structures and attribute information evolve highly rapidly in dynamic graphs, which requires computing the updated PageRank-based proximity matrices and applying SVD over the matrices very frequently. This is prohibitively expensive in real-world applications such as real-time recommendations in E-Commerce websites with hundreds of millions of users and billions of products. First, in terms of the computation of PageRank-based proximities, a promising solution is to tailor the recently proposed approach [179] to update personalized PageRanks on dynamic graphs for maintaining reweighted personalized PageRank and node-attribute PageRank matrices of the dynamic graphs. Unfortunately, this approach requires updating and storing all-pair proximities, which is still costly for large-scale dynamic graphs. On the other hand, a simple and straightforward way for updating embeddings is to exploit incremental SVD techniques [119] instead of performing the singular value decomposition on the full updated matrix all over again. However, this is highly challenging due to that a minor change in the original matrix may lead to big differences in the factorized results. Thus, it is non-trivial to devise efficient techniques for the extension of **NRP** and **PANE** to large-scale dynamic graphs. The basic idea of our sketched approach is to convert the SVD into an optimization procedure with random walk samples based on the PageRank-based proximities. This conversion allows us to decompose the computation and factorization in **NRP** and **PANE**, enabling the parallelization of learning embeddings in a distributed environment. However, the optimization procedure would require sampling a large number of random walks to achieve satisfactory result utility. Towards this end, we can generate and store an amount of short random walks as indexes in the pre-processing steps. The reasons for using short

random walks instead of long ones are two-fold. First, although long random walks capture higher-order proximity information, they are more likely to be updated frequently on dynamic graphs since they involve more nodes. Second, we can concatenate short random walks to obtain long random walks before feeding them to the training process but not the other way around. It would be interesting to devise efficient techniques for maintaining the random walk indexes and study the random walk length such that the online concatenation cost and the offline indexing cost are balanced.

Publications

1. **Renchi Yang**, Jieming Shi, Xiaokui Xiao, Yin Yang, Juncheng Liu, Sourav S Bhowmick, “Scaling Attributed Network Embedding to Massive Graphs”, *Proceedings of the VLDB Endowment (PVLDB)*, pp. 37-49, 2021.
2. Jieming Shi, Tianyuan Jin, **Renchi Yang**, Xiaokui Xiao, Yin Yang, “Realtime Index-Free Single Source SimRank Processing on Web-Scale Graphs”, *Proceedings of the VLDB Endowment (PVLDB)*, pp. 966-978, 2020.
3. **Renchi Yang**, Jieming Shi, Xiaokui Xiao, Yin Yang, Sourav S. Bhowmick, “Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank”, *Proceedings of the VLDB Endowment (PVLDB)*, pp. 670-683, 2020.
4. Jieming Shi, **Renchi Yang**, Tianyuan Jin, Xiaokui Xiao, Yin Yang, “Realtime Top-k Personalized PageRank over Large Graphs on GPUs”, *Proceedings of the VLDB Endowment (PVLDB)*, pp. 15-28, 2020.
5. **Renchi Yang**, Xiaokui Xiao, Zhewei Wei, Sourav Bhowmick, Jun Zhao, and Rong-Hua Li, “Efficient Estimation of Heat Kernel PageRank for Local Clustering”, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 1339-1356, 2019.
6. Sibor Wang, **Renchi Yang**, Runhui Wang, Xiaokui Xiao, Zhewei Wei, Wenqing Lin, Yin Yang, Nan Tang, “Efficient Algorithms for Approximate Single-Source Personalized PageRank Queries”, *ACM Transactions on Database Systems (TODS)*, 44(4):18:1-18:37, 2019.

7. Sibowang, **Renchi Yang**, Xiaokui Xiao, Zhewei Wei, and Yin Yang, “FORA: Simple and Effective Approximate Single-Source Personalized PageRank”, *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 505-514, 2017.

References

- [1] S. Abu-El-Haija, B. Perozzi, R. Al-Rfou, and A. A. Alemi, “Watch your step: Learning node embeddings via graph attention,” in *NIPS*, 2018.
- [2] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. J. Smola, “Distributed large-scale natural graph factorization,” in *WWW*, 2013.
- [3] R. Andersen, F. Chung, and K. Lang, “Local graph partitioning using pagerank vectors,” in *FOCS*, 2006.
- [4] R. Andersen and Y. Peres, “Finding sparse cuts locally using evolving sets,” in *STOC*, 2009.
- [5] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova, “Monte carlo methods in pagerank computation: When one iteration is sufficient,” *SIAM Journal on Numerical Analysis*, vol. 45, no. 2, pp. 890–904, 2007.
- [6] H. Avron and L. Horesh, “Community detection using time-dependent personalized pagerank,” in *ICML*, 2015.
- [7] L. Backstrom and J. Leskovec, “Supervised random walks: Predicting and recommending links in social networks,” in *WSDM*, 2011.
- [8] S. Bandyopadhyay, S. V. Vivek, and M. Murty, “Outlier resistant unsupervised deep architectures for attributed network embedding,” in *WSDM*, 2020, pp. 25–33.
- [9] S. Banerjee and P. Lofgren, “Fast bidirectional probability estimation in markov models,” in *NIPS*, 2015.
- [10] B. Bollobas, *Modern Graph Theory*, 1998.

REFERENCES

- [11] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” in *COMPSTAT*, 2010, pp. 177–186.
- [12] H. Cai, V. W. Zheng, and K. C. Chang, “A comprehensive survey of graph embedding: Problems, techniques, and applications,” *TKDE*, 2018.
- [13] S. Cao, W. Lu, and Q. Xu, “Grarep: Learning graph representations with global structural information,” in *CIKM*, 2015.
- [14] —, “Deep neural networks for learning graph representations,” in *AAAI*, 2016.
- [15] Y. Cen, X. Zou, J. Zhang, H. Yang, J. Zhou, and J. Tang, “Representation learning for attributed multiplex heterogeneous network,” in *SIGKDD*, 2019, pp. 1358–1368.
- [16] S. Chang, W. Han, J. Tang, G.-J. Qi, C. C. Aggarwal, and T. S. Huang, “Heterogeneous network embedding via deep architectures,” in *SIGKDD*, 2015, pp. 119–128.
- [17] H. Chen, B. Perozzi, Y. Hu, and S. Skiena, “HARP: hierarchical representation learning for networks,” in *AAAI*, 2018.
- [18] F. Chung, “The heat kernel as the pagerank of a graph,” *PNAS*, 2007.
- [19] —, “A local graph partitioning algorithm using heat kernel pagerank,” *Internet Mathematics*, 2009.
- [20] F. Chung and L. Lu, “Concentration inequalities and martingale inequalities: a survey,” *Internet Mathematics*, 2006.
- [21] F. Chung and O. Simpson, “Computing heat kernel pagerank and a local clustering algorithm,” in *IWOCA*, 2014.
- [22] —, “Computing heat kernel pagerank and a local clustering algorithm,” *arXiv Preprint*, 2015.
- [23] —, “Distributed algorithms for finding local clusters using heat kernel pagerank,” in *WAW*, 2015.

REFERENCES

- [24] K. W. Church and P. Hanks, “Word association norms, mutual information, and lexicography,” *Computational linguistics*, pp. 22–29, 1990.
- [25] A. Clauset, “Finding local community structure in networks,” *Physical Review E*, 2005.
- [26] P. Comon, X. Luciani, and A. L. De Almeida, “Tensor decompositions, alternating least squares and other tales,” *J. Chemom.*, pp. 393–405, 2009.
- [27] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.
- [28] P. Cui, X. Wang, J. Pei, and W. Zhu, “A survey on network embedding,” *TKDE*, 2018.
- [29] W. Cui, Y. Xiao, H. Wang, and W. Wang, “Local search of communities in large graphs,” in *SIGMOD*, 2014.
- [30] Q. Dai, Q. Li, J. Tang, and D. Wang, “Adversarial network embedding,” in *AAAI*, 2018.
- [31] A. Dempster, N. Laird, and D. Rubin, “Maximum likelihood from incomplete data via the *em* algorithm,” *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 39, no. 1, pp. 1–38, 1977.
- [32] P. Diaconis and J. A. Fill, “Strong stationary times via a new form of duality,” *The Annals of Probability*, 1990.
- [33] Y. Dong, N. V. Chawla, and A. Swami, “metapath2vec: Scalable representation learning for heterogeneous networks,” in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 135–144.
- [34] C. Donnat, M. Zitnik, D. Hallac, and J. Leskovec, “Learning structural node embeddings via diffusion wavelets,” in *KDD*, 2018.

REFERENCES

- [35] L. Erdős, A. Knowles, H.-T. Yau, J. Yin *et al.*, “Spectral statistics of erdős–rényi graphs i: local semicircle law,” *The Annals of Probability*, 2013.
- [36] K. et al., “Heat kernel based community detection,” in *KDD*, 2014.
- [37] —, “What is twitter, a social network or a news media?” in *WWW*, 2010.
- [38] L. et al., “Isorankn: spectral methods for global alignment of multiple protein networks,” *Bioinformatics*, 2009.
- [39] —, “Personalized pagerank estimation and search: A bidirectional approach,” in *WSDM*, 2016.
- [40] P. et al., “The pagerank citation ranking: Bringing order to the web.” Stanford InfoLab, Tech. Rep., 1999.
- [41] S. et al., “Parallel local graph clustering,” *VLDB*, 2016.
- [42] —, “Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems,” in *STOC*, 2004.
- [43] T. et al., “Graph partitioning by spectral rounding: Applications in image segmentation and clustering,” in *CVPR*, 2006.
- [44] W. et al., “Community detection in social networks: an in-depth benchmarking study with a procedure-oriented framework,” *VLDB*, 2015.
- [45] —, “Hubppr: effective indexing for approximate personalized pagerank,” *VLDB*, 2016.
- [46] —, “Capacity releasing diffusion for speed and locality,” in *ICML*, 2017.
- [47] —, “Fora: Simple and effective approximate single-source personalized pagerank,” in *KDD*, 2017.
- [48] —, “Topppr: top-k personalized pagerank queries with precision guarantees on large graphs,” in *SIGMOD*, 2018.

REFERENCES

- [49] P. F. Felzenszwalb and D. P. Huttenlocher, “Efficient graph-based image segmentation,” *IJCV*, 2004.
- [50] S. Fortunato, “Community detection in graphs,” *Physics Reports*, 2010.
- [51] H. Gao and H. Huang, “Deep attributed network embedding,” in *IJCAI*, 2018.
- [52] —, “Self-paced network embedding,” in *KDD*, 2018.
- [53] H. Gao, J. Pei, and H. Huang, “Progan: Network embedding via proximity generative adversarial network,” in *KDD*, 2019.
- [54] S. O. Gharan and L. Trevisan, “Approximating the expansion profile and almost optimal local graph clustering,” in *FOCS*, 2012.
- [55] D. F. Gleich, “Pagerank beyond the web,” *SIAM Review*, vol. 57, no. 3, pp. 321–363, 2015.
- [56] G. H. Golub and C. Reinsch, “Singular value decomposition and least squares solutions,” *Linear Algebra*, pp. 134–151, 1971.
- [57] G. H. Golub and C. F. Van Loan, “Matrix computations. 1996,” *Johns Hopkins University, Press, Baltimore, MD, USA*, 1996.
- [58] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.
- [59] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in *NeurIPS*, 2014, pp. 2672–2680.
- [60] A. Grover and J. Leskovec, “node2vec: Scalable feature learning for networks,” in *KDD*, 2016.
- [61] —, “node2vec: Scalable feature learning for networks,” in *KDD*, 2016.
- [62] Y. Gu, Y. Sun, Y. Li, and Y. Yang, “Rare: Social rank regulated large-scale network embedding,” in *WWW*, 2018.

REFERENCES

- [63] —, “Rare: Social rank regulated large-scale network embedding,” in *WWW*, 2018.
- [64] W. Guo, Y. Li, M. Sha, and K.-L. Tan, “Parallel personalized pagerank on dynamic graphs,” *PVLDB*, vol. 11, no. 1, pp. 93–106, 2017.
- [65] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh, “Wtf: The who to follow service at twitter,” in *WWW*, 2013, pp. 505–514.
- [66] Z. Gyöngyi, H. Garcia-Molina, and J. Pedersen, “Combating web spam with trustrank,” in *PVLDB*, 2004.
- [67] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in *NeurIPS*, 2017.
- [68] T. H. Haveliwala, “Topic-sensitive pagerank,” in *WWW*, 2002, pp. 517–526.
- [69] K. He, P. Shi, J. Hopcroft, and D. Bindel, “Local spectral diffusion for robust community detection,” in *KDD Workshop*, 2016.
- [70] K. He, Y. Sun, D. Bindel, J. Hopcroft, and Y. Li, “Detecting overlapping communities from local spectral subspaces,” in *ICDM*, 2015.
- [71] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, pp. 1735–1780, 1997.
- [72] T. Hogg and K. Lerman, “Social dynamics of digg,” *EPJ Data Science*, 2012.
- [73] Y. Hou, H. Chen, C. Li, J. Cheng, and M.-C. Yang, “A representation learning framework for property graphs,” in *KDD*, 2019.
- [74] J. Hu, R. Cheng, Z. Huang, Y. Fang, and S. Luo, “On embedding uncertain graphs,” in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, pp. 157–166.
- [75] X. Huang, J. Li, and X. Hu, “Accelerated attributed network embedding,” in *SDM*, 2017.

REFERENCES

- [76] Z. Huang and N. Mamoulis, “Heterogeneous information network embedding for meta path based proximity,” *arXiv preprint*, 2017.
- [77] K. Järvelin and J. Kekäläinen, “Ir evaluation methods for retrieving highly relevant documents,” in *SIGIR*, 2000.
- [78] G. Jeh and J. Widom, “Scaling personalized web search,” in *WWW*, 2003, pp. 271–279.
- [79] —, “Scaling personalized web search,” in *WWW*, 2003.
- [80] G. Ji, S. He, L. Xu, K. Liu, and J. Zhao, “Knowledge graph embedding via dynamic mapping matrix,” in *Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers)*, 2015, pp. 687–696.
- [81] D. Jin, B. Li, P. Jiao, D. He, and W. Zhang, “Network-specific variational auto-encoder for embedding in attribute networks,” in *IJCAI*, 2019.
- [82] J. Jung, K. Shin, L. Sael, and U. Kang, “Random walk with restart on large graphs using block elimination,” *TODS*, vol. 41, no. 2, pp. 1–43, 2016.
- [83] Kaggle, “Kdd cup,” 2012, <https://www.kaggle.com/c/kddcup2012-track1>.
- [84] J. Kim, H. Park, J.-E. Lee, and U. Kang, “Side: Representation learning in signed directed networks,” in *WWW*, 2018.
- [85] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *ICLR*, 2016.
- [86] —, “Variational graph auto-encoders,” *arXiv preprint*, 2016.
- [87] J. Klicpera, A. Bojchevski, and S. Günnemann, “Personalized embedding propagation: Combining neural networks on graphs with personalized pagerank,” in *ICLR*, 2019.
- [88] Y.-A. Lai, C.-C. Hsu, W. Chen, M.-Y. Yeh, and S.-D. Lin, “Prune: Preserving proximity and global ranking for network embedding,” in *NIPS*, 2017.

REFERENCES

- [89] E. L. Lawler, *Combinatorial optimization: networks and matroids*. Courier Corporation, 2001.
- [90] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich, “PyTorch-BigGraph: A Large-scale Graph Embedding System,” in *SysML*, 2019.
- [91] —, “Pytorch-biggraph: A large-scale graph embedding system,” in *SysML*, 2019.
- [92] J. Leskovec, “Stanford large network dataset collection,” 2009. [Online]. Available: <http://snap.stanford.edu>
- [93] J. Leskovec, K. J. Lang, and M. Mahoney, “Empirical comparison of algorithms for network community detection,” in *WWW*, 2010.
- [94] J. Leskovec and J. J. Mcauley, “Learning to discover social circles in ego networks,” in *NeurIPS*, 2012, pp. 539–547.
- [95] Y. Li, K. He, D. Bindel, and J. E. Hopcroft, “Uncovering the small community structure in large networks: A local spectral approach,” in *WWW*, 2015.
- [96] D. C. Liu, S. Rogers, R. Shiao, D. Kislyuk, K. C. Ma, Z. Zhong, J. Liu, and Y. Jing, “Related pins at pinterest: The evolution of a real-world recommender system,” in *WWW Companion*, 2017, pp. 583–592.
- [97] J. Liu, Z. He, L. Wei, and Y. Huang, “Content to node: Self-translation network embedding,” in *KDD*, 2018.
- [98] X. Liu, T. Murata, K.-S. Kim, C. Kotarasu, and C. Zhuang, “A general view for network embedding as matrix factorization,” in *WSDM*, 2019.
- [99] J. Ma, P. Cui, X. Wang, and W. Zhu, “Hierarchical taxonomy aware network embedding,” in *KDD*, 2018.
- [100] M. W. Mahoney, L. Orecchia, and N. K. Vishnoi, “A local spectral method for graphs: With applications to improving graph partitions and exploring data graphs locally,” *JMLR*, 2012.

REFERENCES

- [101] Z. Meng, S. Liang, H. Bao, and X. Zhang, “Co-embedding attributed networks,” in *WSDM*, 2019.
- [102] Z. Meng, S. Liang, X. Zhang, R. McCreadie, and I. Ounis, “Jointly learning representations of nodes and attributes for attributed networks,” *TOIS*, pp. 1–32, 2020.
- [103] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *NIPS*, 2013.
- [104] C. Musco and C. Musco, “Randomized block krylov methods for stronger and faster approximate singular value decomposition,” in *NeurIPS*, 2015.
- [105] S. A. Myers and J. Leskovec, “The bursty dynamics of the twitter information network,” in *WWW*, 2014.
- [106] H. Nassar, A. R. Benson, and D. F. Gleich, “Pairwise link prediction,” *arXiv preprint arXiv:1907.04503*, 2019.
- [107] L. Orecchia and Z. A. Zhu, “Flow-based algorithms for local graph clustering,” in *SODA*, 2014.
- [108] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu, “Asymmetric transitivity preserving graph embedding,” in *KDD*, 2016.
- [109] S. Pan, R. Hu, G. Long, J. Jiang, L. Yao, and C. Zhang, “Adversarially regularized graph autoencoder for graph embedding,” in *IJCAI*, 2018.
- [110] B. Perozzi, R. Al-Rfou, and S. Skiena, “Deepwalk: online learning of social representations,” in *KDD*, 2014.
- [111] B. Perozzi, R. Al-Rfou, and S. Skiena, “Deepwalk: Online learning of social representations,” in *SIGKDD*, 2014, pp. 701–710.
- [112] B. Perozzi, V. Kulkarni, H. Chen, and S. Skiena, “Don’t walk, skip!: Online learning of multi-scale network embeddings,” in *ASONAM*, 2017.
- [113] P. Pons and M. Latapy, “Computing communities in large networks using random walks,” in *ISCIS*, 2005.

REFERENCES

- [114] J. Qiu, Y. Dong, H. Ma, J. Li, C. Wang, K. Wang, and J. Tang, “Netsmf: Large-scale network embedding as sparse matrix factorization,” in *WWW*, 2019, pp. 1509–1520.
- [115] J. Qiu, Y. Dong, H. Ma, J. Li, K. Wang, and J. Tang, “Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec,” in *WSDM*, 2018, pp. 459–467.
- [116] P. Radivojac, W. T. Cark, T. R. Oron, A. M. Schnoes, T. Wittkop, A. Sokolov, K. Graim, C. Funk, K. Verspoor, and et. al., “A large-scale evaluation of computational protein function prediction,” *Nature methods*, 2013.
- [117] L. F. R. Ribeiro, P. H. P. Saverese, and D. R. Figueiredo, “struc2vec: Learning node representations from structural identity,” in *KDD*, 2017.
- [118] G. Salton and M. J. McGill, “Introduction to modern information retrieval,” 1986.
- [119] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, “Incremental singular value decomposition algorithms for highly scalable recommender systems.” Citeseer.
- [120] S. E. Schaeffer, “Graph clustering,” *Computer Science Review*, 2007.
- [121] N. Sheikh, Z. T. Kefato, and A. Montresor, “A simple approach to attributed graph embedding via enhanced autoencoder,” in *Complex Networks*. Springer, 2019, pp. 797–809.
- [122] M. Shen, 2017, <https://databricks.com/session/random-walks-on-large-scale-graphs-with-apache-spark>.
- [123] C. Shi, B. Hu, W. X. Zhao, and S. Y. Philip, “Heterogeneous information network embedding for recommendation,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 2, pp. 357–370, 2018.
- [124] J. Shi, R. Yang, T. Jin, X. Xiao, and Y. Yang, “Realtime top-k personalized pagerank over large graphs on gpus,” *Proceedings of the VLDB Endowment*, vol. 13, no. 1, pp. 15–28, 2019.

REFERENCES

- [125] K. Shin, J. Jung, S. Lee, and U. Kang, “Bear: Block elimination approach for random walk with restart on large graphs,” in *SIGMOD*, 2015, pp. 1571–1585.
- [126] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B.-J. Hsu, and K. Wang, “An overview of microsoft academic service (mas) and applications,” in *WWW*, 2015, pp. 243–246.
- [127] M. Sozio and A. Gionis, “The community-search problem and how to plan a successful cocktail party,” in *KDD*, 2010.
- [128] D. A. Spielman and S.-H. Teng, “A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning,” *SICOMP*, 2013.
- [129] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, “Line: Large-scale information network embedding,” in *WWW*, 2015, pp. 1067–1077.
- [130] ———, “LINE: large-scale information network embedding,” in *WWW*, 2015.
- [131] L. Tang and H. Liu, “Leveraging social media networks for classification,” *DMKD*, 2011.
- [132] H. Tong, C. Faloutsos, and J.-Y. Pan, “Fast random walk with restart and its applications,” in *ICDM*. IEEE, 2006, pp. 613–622.
- [133] R. Trivedi, B. Sisman, X. L. Dong, C. Faloutsos, J. Ma, and H. Zha, “Linknbed: Multi-graph representation learning with entity linkage,” in *ACL*, 2018.
- [134] A. Tsitsulin, D. Mottin, P. Karras, and E. Müller, “Verse: Versatile graph embeddings from similarity measures,” in *WWW*, 2018.
- [135] C. E. Tsourakakis, J. Pachocki, and M. Mitzenmacher, “Scalable motif-aware graph clustering,” in *WWW*, 2017.
- [136] K. Tu, P. Cui, X. Wang, P. S. Yu, and W. Zhu, “Deep recursive network embedding with regular equivalence,” in *KDD*, 2018.
- [137] N. Veldt, D. Gleich, and M. Mahoney, “A simple and strongly-local flow-based method for cut improvement,” in *ICML*, 2016.

REFERENCES

- [138] N. Veldt, C. Klymko, and D. F. Gleich, “Flow-based local graph clustering with better seed set inclusion,” in *Proceedings of the 2019 SIAM International Conference on Data Mining*. SIAM, 2019, pp. 378–386.
- [139] P. Veličković, W. Fedus, W. L. Hamilton, P. Liò, Y. Bengio, and R. D. Hjelm, “Deep Graph Infomax,” in *ICLR*, 2019.
- [140] K. Voevodski, S.-H. Teng, and Y. Xia, “Finding local communities in protein networks,” *BMC Bioinformatics*, 2009.
- [141] A. J. Walker, “New fast method for generating discrete random numbers with arbitrary frequency distributions,” *Electronics Letters*, 1974.
- [142] D. Wang, P. Cui, and W. Zhu, “Structural deep network embedding,” in *KDD*, 2016.
- [143] H. Wang, J. Wang, J. Wang, M. Zhao, W. Zhang, F. Zhang, X. Xing, and M. Guo, “Graphgan: Graph representation learning with generative adversarial nets,” in *AAAI*, 2018.
- [144] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, “Billion-scale commodity embedding for e-commerce recommendation in alibaba,” in *KDD*, 2018.
- [145] Q. Wang, S. Wang, M. Gong, and Y. Wu, “Feature hashing for network representation learning.” in *IJCAI*, 2018.
- [146] Q. Wang, Z. Mao, B. Wang, and L. Guo, “Knowledge graph embedding: A survey of approaches and applications,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 12, pp. 2724–2743, 2017.
- [147] R. Wang, S. Wang, and X. Zhou, “Parallelizing approximate single-source personalized pagerank queries on shared memory,” *VLDBJ*, vol. 28, no. 6, pp. 923–940, 2019.
- [148] S. Wang, Y. Tang, X. Xiao, Y. Yang, and Z. Li, “Hubppr: Effective indexing for approximate personalized pagerank,” *PVLDB*, 2016.

REFERENCES

- [149] S. Wang, R. Yang, R. Wang, X. Xiao, Z. Wei, W. Lin, Y. Yang, and N. Tang, “Efficient algorithms for approximate single-source personalized pagerank queries,” *TODS*, vol. 44, no. 4, pp. 1–37, 2019.
- [150] S. Wang, R. Yang, X. Xiao, Z. Wei, and Y. Yang, “FORA: simple and effective approximate single-source personalized pagerank,” in *KDD*, 2017.
- [151] S. Wang, J. Tang, C. Aggarwal, Y. Chang, and H. Liu, “Signed network embedding in social media,” in *Proceedings of the 2017 SIAM international conference on data mining*. SIAM, 2017, pp. 327–335.
- [152] X. Wang, P. Cui, J. Wang, J. Pei, W. Zhu, and S. Yang, “Community preserving network embedding,” in *AAAI*, 2017.
- [153] Z. Wang, J. Zhang, J. Feng, and Z. Chen, “Knowledge graph embedding by translating on hyperplanes.” in *Aaai*, vol. 14, no. 2014, 2014, pp. 1112–1119.
- [154] W. Wei, Y. Wang, P. Gao, S. Sun, and D. Yu, “A distributed multi-gpu system for large-scale node embedding at tencent,” *arXiv preprint arXiv:2005.13789*, 2020.
- [155] Z. Wei, X. He, X. Xiao, S. Wang, S. Shang, and J. Wen, “Topprr: Top-k personalized pagerank queries with precision guarantees on large graphs,” in *SIGMOD*, 2018.
- [156] S. J. Wright, “Coordinate descent algorithms,” *Mathematical Programming*, pp. 3–34, 2015.
- [157] J. Wu and J. He, “Scalable manifold-regularized attributed network embedding via maximum mean discrepancy,” in *CIKM*, 2019, pp. 2101–2104.
- [158] L. Y. Wu, A. Fisch, S. Chopra, K. Adams, A. Bordes, and J. Weston, “Starspace: Embed all the things!” in *AAAI*, 2018.
- [159] W. Wu, B. Li, L. Chen, and C. Zhang, “Efficient attributed network embedding via recursive randomized hashing.” in *IJCAI*, 2018.

REFERENCES

- [160] C. Yang, L. Zhong, L.-J. Li, and L. Jie, “Bi-directional joint inference for user links and attributes on large social graphs,” in *WWW*, 2017.
- [161] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Chang, “Network representation learning with rich text information,” in *IJCAI*, 2015.
- [162] C. Yang, M. Sun, Z. Liu, and C. Tu, “Fast network embedding enhancement via high order proximity approximation,” in *IJCAI*, 2017.
- [163] H. Yang, S. Pan, L. Chen, C. Zhou, and P. Zhang, “Low-bit quantization for attributed network representation learning,” in *IJCAI*, 2019.
- [164] H. Yang, S. Pan, P. Zhang, L. Chen, D. Lian, and C. Zhang, “Binarized attributed network embedding,” in *ICDM*, 2018.
- [165] J. Yang and J. Leskovec, “Defining and evaluating network communities based on ground-truth,” in *KDD Workshop*, 2012.
- [166] ———, “Defining and evaluating network communities based on ground-truth,” *KAIS*, 2015.
- [167] J. Yang, J. McAuley, and J. Leskovec, “Community detection in networks with node attributes,” in *ICDM*, 2013.
- [168] R. Yang, J. Shi, X. Xiao, Y. Yang, and S. S. Bhowmick, “Homogeneous network embedding for massive graphs via reweighted personalized pagerank,” in *PVLDB*, 2020, pp. 670–683.
- [169] R. Yang, J. Shi, X. Xiao, Y. Yang, J. Liu, and S. S. Bhowmick, “Scaling attributed network embedding to massive graphs,” *Proceedings of the VLDB Endowment*, vol. 14, no. 1, pp. 37–49, 2020.
- [170] R. Yang, X. Xiao, Z. Wei, S. S. Bhowmick, J. Zhao, and R.-H. Li, “Efficient estimation of heat kernel pagerank for local clustering,” in *SIGMOD*, 2019, pp. 1339–1356.

REFERENCES

- [171] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich, “Local higher-order graph clustering,” in *KDD*, 2017.
- [172] Y. Yin and Z. Wei, “Scalable graph embeddings via sparse transpose proximities,” in *KDD*, 2019, pp. 1429–1437.
- [173] M. Yoon, W. Jin, and U. Kang, “Fast and accurate random walk with restart on dynamic graphs with guarantees,” in *WWW*, 2018, pp. 409–418.
- [174] M. Yoon, J. Jung, and U. Kang, “Tpa: Fast, scalable, and accurate method for approximate random walk with restart on billion scale graphs,” in *ICDE*, 2018, pp. 1132–1143.
- [175] W. Yu, C. Zheng, W. Cheng, C. C. Aggarwal, D. Song, B. Zong, H. Chen, and W. Wang, “Learning deep network representations with adversarially regularized autoencoders,” in *KDD*, 2018.
- [176] S. Yuan, X. Wu, and Y. Xiang, “Sne: signed network embedding,” in *Pacific-Asia conference on knowledge discovery and data mining*. Springer, 2017, pp. 183–195.
- [177] D. Zhang, J. Yin, X. Zhu, and C. Zhang, “Homophily, structure, and content augmented network representation learning,” in *ICDM*, 2016.
- [178] ———, “Network representation learning: A survey,” *IEEE Trans. Big Data*, 2018.
- [179] H. Zhang, P. Lofgren, and A. Goel, “Approximate personalized pagerank on dynamic graphs,” in *KDD*, 2016.
- [180] J. Zhang, Y. Dong, Y. Wang, J. Tang, and M. Ding, “Prone: Fast and scalable network representation learning,” in *IJCAI*, 2019, pp. 4278–4284.
- [181] Z. Zhang, H. Yang, J. Bu, S. Zhou, P. Yu, J. Zhang, M. Ester, and C. Wang, “Anrl: Attributed network representation learning via deep neural networks.” in *IJCAI*, 2018.
- [182] Z. Zhang, P. Cui, H. Li, X. Wang, and W. Zhu, “Billion-scale network embedding with iterative random projection,” in *ICDM*, 2018.

REFERENCES

- [183] Z. Zhang, P. Cui, X. Wang, J. Pei, X. Yao, and W. Zhu, “Arbitrary-order proximity preserved network embedding,” in *KDD*, 2018.
- [184] C. Zhou, Y. Liu, X. Liu, Z. Liu, and J. Gao, “Scalable graph embedding for asymmetric proximity,” in *AAAI*, 2017.
- [185] —, “Scalable graph embedding for asymmetric proximity,” in *AAAI*, 2017.
- [186] D. Zhou, S. Zhang, M. Y. Yildirim, S. Alcorn, H. Tong, H. Davulcu, and J. He, “A local algorithm for structure-preserving graph cut,” in *KDD*, 2017.
- [187] S. Zhou, H. Yang, X. Wang, J. Bu, M. Ester, P. Yu, J. Zhang, and C. Wang, “Prre: Personalized relation ranking embedding for attributed networks,” in *CIKM*, 2018.
- [188] D. Zhu, P. Cui, D. Wang, and W. Zhu, “Deep variational network embedding in wasserstein space,” in *KDD*, 2018.
- [189] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, “Aligraph: a comprehensive graph neural network platform,” *PVLDB*, pp. 2094–2105, 2019.
- [190] Z. A. Zhu, S. Lattanzi, and V. S. Mirrokni, “A local algorithm for finding well-connected clusters,” in *ICML*, 2013.
- [191] Z. Zhu, S. Xu, M. Qu, and J. Tang, “Graphy: A high performance cpu-gpu hybrid system for node embedding,” in *WWW*, 2019.