universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Graph Compression Techniques for Personalized PageRank

verfasst von | submitted by

Ole Yannick Schlüter BSc.

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von | Supervisor:	Ass.-Prof. Dr.techn. Gramoz Goranci M.Sc.

Acknowledgements

I thank everyone who supported me over the past few months!

I used *ChatGPT* to improve the clarity and language of this thesis manuscript during its preparation. I subsequently reviewed and revised all content as necessary. Although the tool supported paraphrasing and identifying grammatical issues, all material is either original or derived from properly cited sources.

Abstract

The growing size of graph-structured data has made polynomial-time algorithms increasingly impractical; yet many graph problems admit only such algorithms. One approach addressing this challenge is a reduction paradigm known as *graph sparsification* or *graph compression*, a preprocessing technique that reduces the number of edges or nodes in the graph while preserving some graph property of interest. The problem can then be solved on the smaller, preprocessed graph (the *sparsifier*), and the result is used to infer a solution for the original input graph. The quality of the solution depends on how well the sparsifier preserves the properties of the original graph, and the properties to preserve depend on the specific problem domain.

In this work, we investigate *vertex sparsifiers* that preserve *personalized PageRank* values, building on earlier work in this area. We show that constructing such sparsifiers can be reduced to operations on the *Laplacian* matrix of a graph. Leveraging this connection between graphs and matrices, we propose a new algorithm that constructs the same (exact) sparsifiers as existing approaches, but provides a correctness proof based on linear algebra, using the *Schur complement* as the central tool.

This new connection not only simplifies the theoretical analysis but also leads to practical improvements. We develop efficient implementations for constructing exact personalized PageRank sparsifiers that outperform the existing approach on undirected graphs, often achieving at least $2\times$ speedups in practice, and that scale well to large sparse graphs with millions of edges. Furthermore, we introduce an approximation scheme based on approximating the Schur complement, which yields a near-linear running time for constructing sparsifiers and delivers competitive or superior run times across all tested datasets. Regarding the quality of the approximate sparsifiers, our experiments show that they achieve low average error in personalized PageRank values. We further demonstrate that they improve community structure preservation compared to simple induced subgraphs, although the overall quality varies across datasets and remains moderate. While we do not provide formal quality guarantees for the approximation, we also establish a related theoretical limitation for approximating PageRank using sparse graphs.

Kurzfassung

Die zunehmende Größe graphstrukturierter Daten macht Algorithmen mit polynomieller Laufzeit zunehmend unpraktikabel, dennoch lassen sich viele Graphprobleme nur mit solchen Algorithmen lösen. Ein möglicher Ansatz zur Bewältigung dieser Herausforderung ist ein Reduktionsparadigma, das als *Graph-Sparsifizierung* oder *Graph-Kompression* bekannt ist. Dabei handelt es sich um eine Vorverarbeitungstechnik, die die Anzahl der Kanten oder Knoten im Graphen reduziert, während bestimmte relevante Graph-Eigenschaften erhalten bleiben. Das Problem kann auf dem kleineren, vorverarbeiteten Graphen (dem sogenannten *Sparsifier*) gelöst werden. Das Ergebnis dient dazu, eine Lösung für den ursprünglichen Eingabegraphen abzuleiten. Die Qualität der Lösung hängt davon ab, wie genau die Eigenschaften des Originalgraphen im Sparsifier beibehalten werden. Welche Eigenschaften erhalten bleiben sollen, variiert je nach spezifischem Anwendungsbereich.

In dieser Arbeit untersuchen wir *Vertex-Sparsifier*, die *personalisierte PageRank*-Werte bewahren und knüpfen dabei an frühere Arbeiten in diesem Bereich an. Wir zeigen, dass sich die Konstruktion solcher Sparsifier auf Operationen an der *Laplace*-Matrix eines Graphen reduzieren lässt. Durch die Nutzung dieser Verbindung zwischen Graphen und Matrizen entwerfen wir einen neuen Algorithmus, der dieselben (exakten) Sparsifier wie in bestehenden Ansätzen konstruiert, dessen Korrektheit jedoch auf linearer Algebra basiert und das *Schur-Komplement* als zentrales Werkzeug nutzt.

Diese neue Verbindung vereinfacht nicht nur die theoretische Analyse, sondern führt auch zu praktischen Verbesserungen. Wir entwickeln effiziente Implementierungen zur Konstruktion exakter personalisierter PageRank-Sparsifier, die auf ungerichteten Graphen bessere Ergebnisse liefern als der bestehende Ansatz. In der Praxis können wir oft eine mindestens doppelt so schnelle Ausführungszeit beobachten und eine Skalierung auf große Graphen (mit mehreren Millionen Kanten) ermöglichen. Darüber hinaus stellen wir ein Approximationsverfahren vor, das auf der Approximation des Schur-Komplements basiert. Dieses Verfahren erlaubt eine nahezu lineare Laufzeit für die Konstruktion von Sparsifiern und erzielt über alle getesteten Datensätze hinweg eine konkurrenzfähige oder überlegene Ausführungszeit. Bezüglich der Qualität der approximativen Sparsifier zeigen unsere Experimente, dass sie einen geringen durchschnittlichen Fehler für die personalisierten PageRank-Werte aufweisen. Zudem verbessern sie die Erhaltung der Community-Struktur im Vergleich zu einfachen induzierten Teilgraphen, auch wenn die Gesamtqualität zwischen den Datensätzen variiert und moderat bleibt. Wir können keine formalen Qualitätsgarantien für die von uns verwendeten Approximation geben, zeigen dafür aber eine theoretische Limitierung für die Approximation von PageRank mit dünnbesetzten Graphen in einem leicht abgewandelten Rahmen auf.

Contents

List of Tables

List of Figures

List of Algorithms

1. Introduction

Graphs are one of the most essential data structures in computer science, enabling the modeling of relationships and dependencies between objects and in that way serve as models to solve optimization and data analysis problems across a wide variety of areas. Examples include the web graph, social networks, street networks and many more. Over the years, the size of the data, including graph-structured data, has grown due to ongoing technological advancements and digitalization. More recently this growth has been accelerated by the success of AI and machine learning, which rely on large datasets for training. Today, we often encounter significantly larger graphs, making it impractical to apply non-linear time algorithms that depend on the number of nodes and/or edges. Unfortunately, for some problems no efficient algorithms are known. As a result, research focused on approximation and randomized algorithms which offer a better running time at the cost of losing precision or by only providing probabilistic guarantees.

An alternative approach to improving performance is to preprocess the input graph to reduce its size. This reduction paradigm, known as *graph compression* or *graph sparsification*, provides a different approach to address the inherent problem. At a high-level, graph sparsification entails:

1. compress large input graph into a smaller-sized one, called sparsifier;
2. run computationally "expensive" algorithm on the compressed small graph;
3. use the result to infer information about the original input graph.

For example, consider a road network where nodes are cities, edges are connections between cities, and a connection can either be a high or low speed road. We are interested in finding the fastest route from city A to B. One way to reduce the input size is to restrict the network to high-speed connections. This reduces the number of edges of the input and might still give a reasonable approximation for the travel time as we would like to avoid the slow connections for most of the time. Since we reduce the number of edges this is known as an *edge sparsifier*.

In real-world applications, many graphs are already sparse, meaning the number of edges is significantly smaller than the possible number of edges. Alternatively we could reduce the number of nodes and only consider "big" cities. We can assume that big cities are central hubs of our connection network and thus that the reduced network will still give us a good approximation for most connections. The result of this type of graph compression is referred to as a *vertex sparsifier*.

Of course we might be unlucky and split our network into disconnected components, meaning there is no longer a connection between A and B anymore. Therefore we would

like to find good sparsifiers, meaning the compressed graph preserves properties like connectivity, distances or some other useful notion. Another requirement is that the sparsifier can itself be computed efficiently.

1.1. Related Work

Graph sparsification has long been an important research topic. The goal of sparsification is not only to obtain a sparse graph but also to ensure that it preserves certain properties. The property to choose depends highly on the application. Over the years, numerous sparsification techniques have appeared in the literature. In the following chapter we give an overview of the most important techniques and their use cases.

Before we begin, we briefly note that we assume the reader is familiar with basic graph concepts and mathematical notation in this section, as our goal is to provide a concise overview of related topics. All concepts required for the main results of this work will be revisited and explained in detail in Chapter 2.

1.1.1. Edge Sparsifiers

In edge sparsification one is interested in finding a sparse version of a potentially dense input graph. Sparse refers to a graph where the number of edges is significantly lower than the maximum possible $O(n^2)$ edges, ideally scaling asymptotically linearly with the number of nodes. Recall that most real-world graphs are naturally sparse. However, edge sparsification remains relevant because "many algorithms involve the construction of dense graphs, even when solving problems on sparse graphs"[BSST13]. This necessity has led to the development of various edge sparsification techniques.

Spanners. In the introductory example, we observed that it can be useful to preserve distances. Edge sparsifiers that approximate (pairwise) distances are called *spanners*. Formally, given a graph $G = (V, E)$, a subgraph $H = (V, E')$ is called a t-spanner of G if

$$\forall u, v \in V : dist_H(u, v) \leq t \cdot dist_G(u, v).$$

For a pair of nodes $u, v \in V$, the fraction $\frac{dist_H(u,v)}{dist_G(u,v)}$ is called the *stretch* of u, v and $t = \max_{u,v \in V} \frac{dist_H(u,v)}{dist_G(u,v)}$ is the stretch of H. The concept of t-spanners was first properly defined by [PS89]. Later, the authors of [ADD$^+$93] improved and generalized the work to weighted graphs. They proved that for every (weighted) graph, there exists a $(2t + 1)$-spanner with $O(n^{1+1/t})$ edges, and presented an algorithm to compute such a spanner.

It is worth noting that there is an inherent trade-off in the construction of spanners between *quality*, how well distances are approximated, and *sparsity*, how many edges the sparsifier contains. This trade-off is not unique to spanners but applies to most sparsifiers, requiring careful balancing.

Spanners have both practical and theoretical applications. In distributed systems spanners can be used to reduce the resources needed for communication. Instead of

broadcasting the message using the entire system one can use a spanner to reduce the number of connections while maintaining similar communication speeds. A theoretical application is in the context of approximation algorithms, e.g. solving the traveling salesperson problem. For a comprehensive overview on spanners, we refer the reader to [ABS+20].

Cut Sparsifiers. A *cut* of graph $G = (V, E, w)$ is a partition of the vertex set V into two disjoint subsets S and $V \setminus S$. The value of the cut is the sum of the weights of edges crossing the cut, $\text{cut}_G(S) = \sum_{u \in S, v \in V \setminus S} w(u, v)$. In [BK96] the authors introduced the concept of *cut sparsifiers*, which are sparse subgraphs that maintain all values of all cuts. Formally, given a graph $G(V, E, w)$, a subgraph $H(V, E', w')$ is a quality q-cut-sparsifier if

$$\forall S \subset V : \text{cut}_G(S) \leq \text{cut}_H(S) \leq q \cdot \text{cut}_G(S)$$

The authors of [BK96] also presented an algorithm to compute a quality $(1 + \epsilon)$-cut-sparsifier with $O(n \log(n)/\epsilon^2)$ edges in $O(m \log^3(n))$ time. The running time has been reduced by [FHHP19] to $O(m \log^2(n))$. The algorithm can be used to solve cut problems that depend on the number of edges. For example, it can be applied to the push-relabel algorithm to find an s-t minimum cut, achieving a $(1 + \epsilon)$-approximation in $\widetilde{O}(n^2)$ time.

Spectral Sparsifiers. A graph $G(V, E, w)$ has several matrix representations. The most well-known representation is probably the adjacency matrix A_G, another is the *Laplacian* matrix L_G which is defined as:

$$L_G(u, v) = \begin{cases} -w(u, v) & \text{if } u \neq v, \\ \sum_{(u,z) \in E} w(u, z) & \text{otherwise.} \end{cases}$$

The Laplacian is a fundamental construction in spectral graph theory, a field that studies the properties of a graph in relation to the eigenvalues and eigenvectors of the graph's matrix.

In [ST04] the *quadratic form* of the Laplacian $\mathbf{x}^T L_G \mathbf{x} = \sum_{(u,v) \in E} w(u, v)(\mathbf{x}(u) - \mathbf{x}(v))^2$ for some $\mathbf{x} \in \mathbb{R}^{|V|}$ is used to define spectral similarity between graphs. We say that, given a graph $G = (V, E, w)$, a graph $H = (V, E', w')$ is a *spectral sparsifier* of quality q if

$$\forall \mathbf{x} \in \mathbb{R}^{|V|} : \quad \mathbf{x}^T L_G \mathbf{x} \leq \mathbf{x}^T L_H \mathbf{x} \leq q \cdot \mathbf{x}^T L_G \mathbf{x}.$$

Cut and spectral sparsification are closely related. If we restrict $\mathbf{x} \in \{0, 1\}^{|V|}$, the definition is equivalent to that of cut sparsifiers. Thus every spectral sparsifier is a cut sparsifier and spectral similarity is a stronger notion as for example the n-path and n-cycle graphs are 2 cut similar but only n spectrally similar.

In their initial work [ST04], the authors showed that for every weighted undirected graph, a spectral sparsifier with $O(\frac{n}{\epsilon^2} \log^c(\frac{n}{\epsilon}))$ edges and quality $(1 + \epsilon)$ exists, where c is some large constant. In [SS11] the authors improved their previous results by constructing sparsifiers of the same quality but with only $O(n \log(n)/\epsilon^2)$ edges.

One motivation behind spectral sparsifiers lies in solving linear systems of equations $A\mathbf{x} = \mathbf{b}$ efficiently. For general matrices, there is no algorithm that runs in near-linear time in the number of nonzero entries. Using spectral sparsifiers, the authors of [ST04] achieved this for symmetric and diagonally dominant (SDD) systems. SDD systems can be reduced to Laplacian systems [Gre96], and the connection between Laplacian matrices and graphs enables us to use spectral sparsifiers as preconditioners to speed up existing iterative system solvers. Since the initial work in [ST04], SDD system solvers have been continuously improved [KMP11, CKM$^+$14].

1.1.2. Vertex Sparsifiers

Consider the case where the input graph $G = (V, E)$ is already sparse or the focus is on a small subset of nodes $K \subseteq V$. In such scenarios, edge sparsification is not effective, either because the number of edges is already small or because we want the runtime of algorithms to depend only on $|K|$ rather than $|V|$. These observations motivate the work on vertex sparsifiers. Similar to edge sparsifiers, there are various types of vertex sparsification which we will briefly summarize here.

Before presenting related work on vertex sparsifiers, we start with a closely related topic.

Graph Sampling. Sampling a representative subset from a large population is a common technique in fields like data analysis and can also be applied to graph-structured data. Using a sampling strategy, one can select a subset of nodes of the desired size to form the so-called terminal set K. Typically, the induced subgraph on these nodes is then taken as the sampled graph.

In [LF06], various sampling strategies are presented, evaluated, and compared, showing that the choice of sampling method has a significant impact on how well specific properties of the input graph are preserved in the induced subgraph. This highlights that even the way we select the terminal nodes can play a crucial role in vertex sparsification and is an important consideration when constructing vertex sparsifiers. However, depending on the application, the induced subgraph may be a relatively weak approximation, potentially failing to preserve even basic properties like connectivity and offering no guarantees.

In the context of vertex sparsification, we typically assume that a subset of vertices K is already given, and the goal is to construct a graph $H = (K, E')$ that preserves properties of the original graph with some formal guarantees. The nodes in K are called *terminals*, and their count is denoted by $k = |K|$.

Cut and Flow Sparsifiers. Similar to edge sparsifiers that preserve cuts, [Moi09] introduces the concept of *vertex cut sparsifiers*. Intuitively, given a subset of terminals $U \subseteq K$, a vertex cut sparsifier approximately preserves the value of the minimum cut that separates U from $K \setminus U$ in both the original graph G and the sparsifier H. Formally, we define the minimum cut separating U and $K \setminus U$ in G as

$$\mathrm{cut}'_G(U) = \min_{A \subseteq V \, s.t. \, A \cap K = U} \mathrm{cut}_G(A).$$

A graph $H(K, E', w')$ is a *quality* q cut sparsifier (with $q \geq 1$) if

$$\forall U \subset K: \quad \text{cut}'_G(U) \leq \text{cut}_H(U) \leq q \cdot \text{cut}'_G(U).$$

Due to the duality between cuts and flows, cut sparsifiers also approximately preserve s-t flows. In s-t flow problems, the objective is to route (sufficient) flow from a single source node s to a single target node t. The concept of cut sparsifiers was later extended by [LM10] to *flow sparsifiers*, which approximate multicommodity flows.

In the multicommodity flow problem, we are given a capacitated graph $G(V, E, c)$ and a set of demands $d = \{(s_i, t_i, d_i)\}$, where $d_i > 0$ specifies the amount of flow to be routed from source node s_i to target node t_i. A multicommodity flow f can be defined as assigning a flow f_i to each demand (s_i, t_i, d_i). Given such a flow f, the congestion of an edge e is defined as $\sum_i f_i(e)/c(e)$, where $f_i(e)$ is the amount of flow routed over edge e, and $c(e)$ is the capacity of edge e. The congestion of the flow f is the maximum edge congestion and the congestion of demand d, denoted $cong_G(d)$, is the minimum congestion over all feasible flows satisfying d. Given a graph $G = (V, E, c)$ with a terminal set $K \subseteq V$, we say that a graph $H(K, E', c')$ is a quality q flow sparsifier of G if, for every demand set d with demands only between terminals, it holds that

$$cong_H(d) \leq cong_G(d) \leq q \cdot cong_H(d).$$

In [BK96] the authors proved the existence of $O(\frac{\log(k)}{\log(\log(k))})$-quality sparsifiers for general graphs. Subsequently, multiple researchers independently proposed algorithms to construct sparsifiers of this quality [MM10, CLLM10, EGK$^+$14]. For many simple graphs, even constant quality sparsifiers exist. The best-known general lower bound for cut sparsifiers is $\Omega(\frac{\sqrt{\log(k)}}{\log(\log(k))})$ while for flow sparsifier it is $\Omega(\sqrt{\frac{\log(k)}{\log(\log(k))}})$, as established in [MM10].

Cut and flow sparsifiers have become valuable tools in many approximation algorithms because their quality depends only polylogarithmically on k rather than the total number of nodes n. Applications include improved approximations for the requirement cut problem and the development of an approximation algorithm for the Steiner Minimum Linear Arrangement Problem.

Cut and Flow Sparsifier with Steiner Nodes. The sparsifiers discussed above consist only of the terminal set K. If we loosen this requirement and allow additional nodes in the sparsifier, referred to as *Steiner nodes*, can we achieve better approximation guarantees? For exact sparsifiers, also known as mimicking networks, there is a known upper bound of $O(2^{2^k})$ nodes ([HKNR98]) and a lower bound of $\Omega(2^k)$ nodes [KR13, KR14] when preserving cuts. These bounds imply that, in general, sparsifiers for arbitrary graphs may require a number of nodes exponential in k. For constant-quality sparsifiers, [Chu12] demonstrated the existence of cut sparsifiers of size $O(C^3)$ and flow sparsifiers of size $C^{O(\log(\log(C))}$, where C is the sum of capacities incident to the terminal nodes.

Constructing cut and flow sparsifiers typically requires polynomial time in n. This raises the question of whether subpolynomial-time algorithms can be designed that produce sparsifiers with slightly weaker approximation guarantees. An initial answer is

provided by [RST14], where the authors showed how to construct a tree, that contains at most k additional Steiner nodes and is a quality $O(\log^2(n)\log^2(k))$-sparsifier. Due to advancements in computing an approximate max flow, such trees can be computed in near-linear time. However, this improvement in runtime comes at the cost of the quality depending on n.

Approximate Schur Complements. The Schur complement is an important matrix construction that can also be used in graph sparsification by decreasing the size of graph associated matrices. Let $M = \begin{bmatrix} M_{S,S} & M_{S,B} \\ M_{B,S} & M_{B,B} \end{bmatrix}$ be a square matrix divided into four blocks according to disjoint index sets S and B. The Schur complement of M on B is defined as:

$$SC_B(M) = M_{B,B} - M_{B,S}M_{S,S}^{-1}M_{S,B}$$

This operation arises naturally from block Gaussian elimination, where the rows and columns corresponding to S are eliminated. In the context of graph theory, the Schur complement becomes particularly useful when M is the Laplacian matrix L_G of a graph $G = (V, E)$, and $S \cup B = V$ represents a partition of the node set. In this case, $SC_B(L_G)$ is a Laplacian matrix of a reduced graph on the vertex set B, which preserves effective resistances between nodes in B.

However, computing the Schur complement exactly is often computationally expensive, as it typically yields dense matrices even when the input graph is sparse. Because of its connection to Gaussian elimination, early approximation techniques were developed in the context of linear system solvers [KS16], where complete elimination was applied. Building on this, [DKP$^+$17] introduced the notion of approximate Schur complements for partial elimination with respect to a designated terminal set K.

Formally, given an a graph $G = (V, E, w)$ and set of terminal nodes $K \subset V$, a graph $H = (K, E', w')$ is called $(1 \pm \epsilon)$-approximate Schur complement if

$$\forall \mathbf{x} \in \mathbb{R}^{|K|}: \quad (1-\epsilon)\mathbf{x}^T SC_K(L_G)\mathbf{x} \leq \mathbf{x}^T L_H \mathbf{x} \leq (1+\epsilon)\mathbf{x}^T SC_K(L_G)\mathbf{x}$$

The authors of [DKP$^+$17] showed how to compute such an approximate Schur complement with probability at least $1-\gamma$ in time $\widetilde{O}\left(m\log^3\left(\frac{n}{\gamma}\right) + n\varepsilon^{-2}\log^4\left(\frac{n}{\gamma}\right)\right)$ with $O(k\varepsilon^{-2}\log(n/\gamma))$ edges.

[DKP$^+$17] used this construction to improve the running time for random tree sampling. The authors of [GHP18] noted, that approximate Schur complements can be viewed as effective resistance vertex sparsifiers and applied them to the problem of dynamically maintaining all pairs effective resistances.

PageRank Sparsifiers. In [VCG11], the authors study whether it is possible to construct vertex sparsifiers that preserve PageRank values $\mathbf{p}^G$. PageRank, initially introduced by [PBMW99], is a node-ranking algorithm that depends on two parameters: the restart probability $\alpha \in [0, 1]$ and the personalization vector $\mathbf{r} \in \{\mathbf{x} \in \mathbb{R}_+^{|V|} || \mathbf{x}| = 1\}$. When $\mathbf{r}(v) = 1$ for some node $v \in V$ and zero everywhere else, this yields the so-called

personalized PageRank $\mathbf{p}_v^G$. We defer a detailed discussion of PageRank to Section 2.2, where its computation and properties are explained more thoroughly.

We now formally define personalized PageRank sparsifiers: Given a graph $G = (V, E, w)$, we define $H = (K, E', w')$ to be a *quality* ($\gamma\delta$) personalized PageRank (PPR) sparsifier for $\gamma, \delta \geq 1$, if the following condition holds:

$$\forall u, v \in K: \quad \frac{1}{\delta} \cdot \frac{\mathbf{p}_u^G(v)}{\sum_{w \in V} \mathbf{p}_u^G(w)} \leq \frac{\mathbf{p}_u^H(v)}{\sum_{w \in K} \mathbf{p}_u^H(w)} \leq \gamma \cdot \frac{\mathbf{p}_u^G(v)}{\sum_{w \in V} \mathbf{p}_u^G(w)}$$

Here, the additional normalization is necessary because the PageRank vector $\mathbf{p}$ is a probability distribution that sums to 1.

The authors of [VCG11] showed that it is impossible to construct a quality 1-sparsifier with exactly k nodes. However, they demonstrated that a quality 1-sparsifier (exact sparsifier) can be achieved by adding just a single additional node. In the special case where $G(A \cup B, E, w)$ is bipartite, the authors in [EFL+14] showed how to construct an exact sparsifier on just A or B, without adding any extra nodes, simply by reweighting existing edges and adjusting the restart probability α.

The correctness proofs of both constructions rely on preserving random walks within the sparsifier. For example, consider a graph G with three nodes a, b, c and a walk $a \to b \to c$ with some probability p. A sparsifier H that contains only a and c, but not b, should preserve the walk $a \to c$ with the same probability p. This becomes significantly more intricate when dealing with more complex graph structures (beyond bipartite graphs) or when accounting for the restart parameter α. The proofs require a careful mapping of walks from G to H, which leads to highly specialized and technically involved constructions.

The authors in [VCG11] present several additional results. By adding a second auxiliary node, the global PageRank (where the personalization vector $\mathbf{r}$ is the uniform vector) can also be preserved exactly. Moreover, since the constructed sparsifiers are typically dense, they propose a simple rounding scheme that removes edges with weights below a threshold τ. For the resulting pruned sparsifier, the average error across a node's personalized PageRank values is bounded by $2\tau\frac{1-\alpha}{\alpha}$. However, this approximation approach still requires computing the exact solution first.

In [EFL+14] sparsifiers are used to speed up similarity computations, for example, to identify similar objects for recommendation tasks, such as finding users with similar interests and suggesting relevant content to them. One application highlighted by the authors in [VCG11] is clustering: when working with subgraphs, PageRank sparsifiers can preserve community structure significantly better than simple induced subgraphs.

1.1.3. Tree-based Sparsification

Trees are algorithmically attractive, as many problems can be solved more efficiently on trees than on general graphs. Thus, an interesting question in graph sparsification is, what approximation bounds can be achieved when restricting sparsifiers to tree structures.

Low Stretch Spanning Trees. Unfortunately, for distance-based sparsifiers, low stretch cannot always be achieved. Consider the cycle C_n with unit edge weights: the stretch of any spanning tree is $\frac{n-1}{1}$, showing that a single tree may not provide good worst-case guarantees. However, by relaxing the goal to minimizing the *average* stretch, [AKPW95] showed that it is possible to construct a spanning tree with average stretch $e^{O(\sqrt{\log(n)\log(\log(n))})}$. Moreover, they demonstrated the existence of a distribution over such trees for which the expected stretch remains bounded by the same quantity. This result was later significantly improved in [EEST08], which constructed spanning trees with average stretch $O(\log^2(n)\log(\log(n)))$.

Probabilistic Tree Embeddings. In [Bar96], the authors study the problem of embedding a metric space, such as the shortest-path distance of a connected, undirected graph $G = (V, E, w)$, into a simpler metric space, specifically a tree metric. Building on the ideas of [AKPW95], which focused on spanning trees, this work generalizes the setting to consider any tree metric on the vertex set V (e.g., trees with additional Steiner nodes). To this end, the authors introduce a structure known as a *hierarchical well-separated tree* (HST). These trees are constructed by recursively partitioning the node set V into smaller subsets until each subset contains only a single node. This hierarchical decomposition naturally defines a tree structure, which can then be used to approximate distances: distances in the original graph are estimated by computing distances in the tree. While certain graphs, such as cycles, cannot be approximated well by a single tree metric, the authors show that a distribution over tree metrics can yield good approximations. Specifically, the paper shows that by sampling from an appropriate distribution over such trees, one obtains a probabilistic embedding with an expected distortion of $O(\log^2(n))$. This bound was later improved in [Bar98] to $O(\log(n)\log(\log(n)))$, and ultimately to $O(\log(n))$ in [FRT04], matching known lower bound. These embeddings enable efficient approximation algorithms for distance-based problems in general graphs by solving them on trees and transferring the results back to the original graph causing only a logarithmic loss in quality.

Tree Cut Sparsifiers. The techniques above work only for distance-based cost functions and do not extend to properties like cuts or flows. For these, the authors of [Räc02] proposed a decomposition method similar to [Bar96], which also yields a tree but one that approximates the graph's cut structure. Their approach guarantees that the congestion of any flow in the decomposed tree is no greater than in the original graph. Moreover, a solution to the multicommodity flow problem found on the tree can be mapped back to the original graph G, with only a bounded increase in congestion determined by the approximation factor. By using a distribution over a set of trees, the authors of [Räc08] improved the approximation factor from $O(\log^3(n))$ to $O(\log(n))$. A key motivation for these tree-based cut sparsifiers is their usefulness in applications such as oblivious routing, where they enable optimal $O(\log(n))$-competitive algorithms.

1.1.4. Sparsification in Directed Graphs

Many of the sparsification techniques discussed above apply only to undirected graphs, as they possess desirable properties, such as equal in-degrees and out-degrees for all vertices. In fact, there are directed graphs that require all edges to preserve all cuts. Despite this limitation, there have been advancements for directed sparsifiers by depending on different similarity notions such as cut balance [CCPS21] or by restricting the input to specific families of directed graphs, such as Eulerian graphs [CKP$^+$17]. An Eulerian graph is a directed graph where each vertex has equal in-degree and out-degree. This additional property enables the construction of cut sparsifiers and has resulted in an almost linear directed system solver [CKP$^+$17][CKK$^+$18].

1.2. Our Work

In this work, we study personalized PageRank sparsifiers and build upon the work introduced by [VCG11], improving both the theoretical and practical aspects. The work is structured in the following way:

- **Chapter 2** provides the necessary background, definitions, and notation. We introduce key lemmas and mathematical tools that will be essential for the development of our main results.
- **Chapter 3** presents our main algorithm for constructing personalized PageRank sparsifiers. We prove its correctness, analyze its performance, and show how approximation techniques can be used to accelerate it. We also derive a theoretical lower bound for the quality of sparse approximations.
- **Chapter 4** focuses on practical implementation and empirical evaluation. We compare the performance of our algorithm with that of [VCG11], assess the quality of the approximate sparsifiers, and apply them to a clustering task to evaluate their effectiveness in practice.
- **Chapter 5** summarizes the results and highlights open problems.

The main contributions are

- **Simplified Proof:** We present a cleaner and more modular proof based on matrix augmentation and the Schur complement and its connection to Gaussian elimination. Unlike the earlier work in [VCG11] that relies on more intricate random walk constructions, our approach uses well-known matrix identities and properties, making the analysis more accessible.
- **Faster Algorithms:** By reducing the problem to computing a Schur complement, a well-studied operation, we can leverage existing efficient heuristics, iterative solvers, and data structures. Furthermore, we demonstrate how symmetries in undirected input graphs can be exploited to accelerate the computation even further.

- **Foundations for Approximate Sparsification:** The relationship between our algorithm and the Schur complement enables the application of existing approximation methods for the Schur complement, leading to a novel approach for constructing approximate personalized PageRank sparsifiers. This leads to a theoretically efficient near-linear algorithm, which, as demonstrated in our experiments, shows potential for practical applicability.
- **Comprehensive Performance Evaluation:** We conduct a comprehensive performance evaluation by benchmarking the runtime and scalability of various implementations. Our findings are not only useful for PageRank sparsification but may also be of interest to other applications that rely on efficiently computing Schur complements. We demonstrate that our method for constructing exact sparsifiers outperforms existing methods by at least a factor of $2\times$ across most datasets.

2. Preliminaries

2.1. Notation and Definitions

2.1.1. Vectors and Matrices

- Matrices are represented by capital letters (e.g., M), vectors by lowercase bold letters (e.g., $\mathbf{x}$), and scalars by lowercase regular letters (e.g., a).
- To access the i-th element of a vector, we use $\mathbf{v}(i)$, and to access the element in row i and column j of a matrix, use $M(i, j)$.
- When accessing multiple rows or columns, we use the notation $M(i, j : k)$ to access elements in row i from column j to column k, and $M(j : k, i)$ to access elements in column i from row j to row k. This notation can also be extended to access submatrices by specifying ranges for both rows and columns.
- To access an entire row or column, we use the common notation $M(i, :)$ for rows and $M(:, j)$ for columns.

The identity matrix is denoted by I, a diagonal matrix given a vector $\mathbf{x}$ is written as $\text{diag}(\mathbf{x})$, the all-zero vector is denoted by $\mathbf{0}$, and the all-ones vector is written as $\mathbf{1}$. The vector $\mathbf{e}_i$ is a standard basis vector that is zero except for $\mathbf{e}_i(i) = 1$.

2.1.2. Graphs

We consider directed weighted graphs $G = (V, E, w)$, where V is the set of vertices, $E \subseteq V \times V$ is the set of directed edges, and $w : E \to \mathbb{R}^+$ is a function assigning positive weights to the edges. Throughout, we assume that the vertices are labeled $\{1, 2, ..., n\}$, allowing us to use them as indices in matrices and vectors. We work with both directed and undirected graphs, and sometimes with unweighted versions as well. Since directed weighted graphs generalize undirected unweighted graphs, we will define most properties in the context of directed weighted graphs.

For a node $v \in V$ in a directed graph $G(V, E, w)$ the *out-neighborhood* is defined as $N_G^{out}(v) = \{u \in V | (v, u) \in E\}$ and the *in-neighborhood* is defined as $N_G^{in}(v) = \{u \in V | (u, v) \in E\}$. The *out-degree* of node v is defined as $\deg_G^{out}(v) = |N_G^{out}(v)|$ and the *weighted out-degree* is defined as $\deg_{w_G}^{out}(v) = \sum_{u \in N_G^{out}(v)} w(v, u)$. Analogously we define the properties for the in-neighborhood, $\deg_G^{in}(v) = |N_G^{in}(v)|$ and $\deg_{w_G}^{in}(v) = \sum_{u \in N_G^{in}(v)} w(u, v)$. For undirected graphs, in- and out-neighborhoods coincide, so we simply write $N_G(v)$, $\deg_G(v)$, and $\deg_{w_G}$ without further qualification.

A *walk* is a sequence of vertices $v_1 \to v_2 \to ... \to v_k$ such that $(v_i, v_{i+1}) \in E$ for all $0 < i < k$, a *path* is a walk in which all vertices are distinct and a *cycle* is a path where the first and last vertex are the same, i.e., $v_1 = v_k$. The *length* of a walk is the number of edges in the walk, $k - 1$.

We can associate various matrices with a graph G. The *adjacency matrix* A_G is defined as:

$$A_G(u, v) = \begin{cases} w(u, v) & \text{if } (u, v) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

The *degree matrix* is a diagonal matrix defined with respect to a specific notion of degree. For example, the weighted out-degree matrix $D^{out}_{w_G}$ is defined by $D^{out}_G(v, v) = \deg^{out}_{w_G}(v)$.

When it is clear from context, we omit the subscript G for simplicity.

2.2. PageRank

PageRank [PBMW99] is an algorithm that computes a ranking over a graph's nodes. Originally developed for search engines, it was used to rank web pages in response to a query. A central challenge in web search is evaluating the quality of content, which can be easily manipulated. To address this, PageRank leverages the link structure of the web, interpreting hyperlinks as "votes" for a page's importance. Pages that receive many such votes are considered more important. However, not all votes are equal. Links from authoritative or well-linked pages carry more weight than links from obscure websites. Therefore, the importance of a page is recursively defined by the importance of the pages linking to it. This intuition forms the basis of PageRank. Formally, let $G(V, E, w)$ be a directed, weighted graph. The PageRank score $\mathbf{p}(v)$ for a node $v \in V$ is defined as

$$\mathbf{p}(v) = \sum_{(u,v)\in E} \frac{w(u, v)}{\deg^{out}_{w_G}(u)} \mathbf{p}(u). \tag{2.2}$$

In matrix notation this becomes

$$\mathbf{p} = A^T D^{-1}_{out} \mathbf{p}. \tag{2.3}$$

The PageRank vector $\mathbf{p}$ is therefore an eigenvector of the matrix $A^T D^{-1}$ corresponding to eigenvalue 1. It can be computed either by solving the eigenvalue problem directly or by solving the equivalent linear system:

$$(I - A^T D^{-1}_{out})\mathbf{p} = \mathbf{0}. \tag{2.4}$$

Alternatively, PageRank can be viewed from the perspective of random walks on a graph. Consider the web (graph) and a random surfer (walker) who starts at some webpage v and follows links to other webpages. At each step, the surfer follows a link (v, u) with probability $w(v, u)/d^{out}(v)$. Let $\mathbf{p_i}$ denote the probability distribution over webpages

after i steps, where $\mathbf{p}_i(u)$ gives the probability that the surfer is on page u at step i. The distribution at the next step is given by

$$\mathbf{p}_{i+1} = A^T D_{out}^{-1} \mathbf{p}_i. \tag{2.5}$$

where A is the weighted adjacency matrix and D is the diagonal matrix of out-degrees of the web graph. This random process defines a *Markov chain* [Law06], since the future state depends only on the current state and not on the sequence of previous states. The corresponding transition matrix is $P = A^T D^{-1}$. When

$$\mathbf{p}_{i+1} = A^T D_{out}^{-1} \mathbf{p}_i = \mathbf{p}_i, \tag{2.6}$$

$\mathbf{p}_i$ is called the *stationary distribution* and again corresponds to the eigenvector with eigenvalue 1. Therefore, the above vote-based interpretation of PageRank and the random walk formulation are equivalent. Using the theory of Markov chains, one can establish conditions under which a stationary distribution exists. We present these conditions in the context of graphs.

Definition 2.2.1. *A directed graph $G(V, E, w)$ is said to be strongly connected if, for every pair of vertices $u, v \in V$, there exists a directed path from u to v.*

Definition 2.2.2. *A directed graph $G(V, E, w)$ is called aperiodic if the greatest common divisor of the lengths of all cycles in the graph is 1.*

Lemma 2.2.1. *If a directed graph $G(V, E, w)$ is strongly connected and aperiodic, then the Markov chain associated with $P = A_G^T D_G^{-1}$ has a unique stationary distribution that is independent of the initial state.*

For a detailed proof, see [Law06].

However, many real-world graphs, such as the web graph, are neither strongly connected nor aperiodic, which are conditions required for the existence of a unique stationary distribution. To overcome these limitations, PageRank is typically extended through a modified formulation that includes two additional parameters: the *restart probability* α and the *personalization vector* $\mathbf{r}$, a stochastic vector over the nodes. In the random walk interpretation, this extension works as follows: at each step, the walker continues with probability $1 - \alpha$ by following an outgoing edge as before, but with probability α, the walker "restarts" and jumps to a node v with probability $\mathbf{r}(v)$. This ensures that the resulting graph is both strongly connected and aperiodic, thus guaranteeing the existence and uniqueness of a stationary distribution. The updated PageRank formula is then given by:

$$\mathbf{p} = \alpha \mathbf{r} + (1 - \alpha) A^T D_{out}^{-1} \mathbf{p} \tag{2.7}$$

Since $\mathbf{p}$ is stochastic, we have $\mathbf{1}^T \mathbf{p} = 1$, and an alternative version of the formula is:

$$\mathbf{p} = (\alpha \mathbf{r} \mathbf{1}^T + (1 - \alpha) A^T D_{out}^{-1}) \mathbf{p}. \tag{2.8}$$

We can interpret this modification as augmenting the original graph G with additional edges and scaling the weights of the existing ones. For each node pair (u, v), a new

restart edge is added with weight $w(u, v) = \alpha \mathbf{r}(v)$, and the weights of the original edges are normalized by the out-degree and scaled by $1 - \alpha$. This construction, illustrated in Figure 2.1, results in a graph where each node has a normalized out-degree, i.e., the rows of the adjacency matrix sum to one. We refer to the resulting augmented graph as the PageRank graph G_{PageRank}, and its (transposed) adjacency matrix is given by $\alpha \mathbf{r}\mathbf{1}^T + (1 - \alpha)A^T D_{\text{out}}^{-1}$.

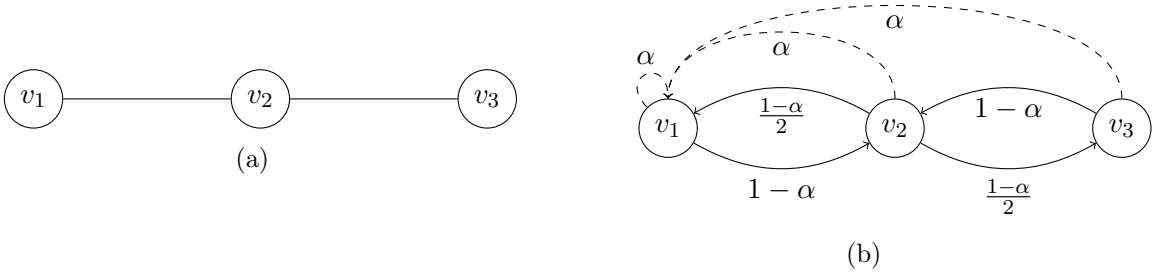

Figure 2.1.: (a) A Graph G, and (b) its corresponding PageRank graph for personalization vector $\mathbf{e}_{v_1}$

To handle nodes with no outgoing edges (also called *dangling nodes*), typically additional outgoing edges are added. These edges are usually chosen to mirror the restart distribution $\mathbf{r}$. This modification ensures that the transition matrix remains stochastic and well-defined.

When $\mathbf{r}$ is the uniform distribution with all entries equal to $1/|V|$, the resulting PageRank vector $\mathbf{p}$ is referred to as the *global PageRank*. However, the personalization vector $\mathbf{r}$ can be chosen arbitrarily to bias the ranking towards specific nodes, for example, to prioritize certain webpages in the web graph. In this work, we focus on preserving the so called *personalized PageRank*, in particular the variant personalized with respect to a single node.

Definition 2.2.3. *(u-personalized PageRank) Given a directed, weighted graph $G(V, E, w)$ and a node $u \in V$, the u-personalized PageRank vector is defined as the unique solution $\mathbf{p}_u$ to the system*

$$\mathbf{p}_u = \alpha \mathbf{r}_u + (1 - \alpha)A^T D_{out}^{-1} \mathbf{p}_u,$$

where $\mathbf{r}_u$ is the unit vector with a 1 in the position corresponding to node u.

2.3. Laplacian Matrix

Given an undirected graph $G = (V, E, w)$ with non-negative edge weights, consider a vector $\mathbf{x} \in \mathbb{R}^{|V|}$. The vector $\mathbf{x}$ can be seen as a function mapping from nodes to numerical values. Matrices associated with the graph, such as its adjacency matrix A_G, can be interpreted as operators transforming $\mathbf{x}$. Specifically, the adjacency matrix operates as follows:

$$(A\mathbf{x})(v) = \sum_{(u,v)} A(u, v)\mathbf{x}(u) \tag{2.9}$$

It acts as an operator propagating information from a node to its adjacent nodes. Another fundamental matrix in this context is the *Laplacian* matrix L_G, which naturally arises from its quadratic form [Spi12]:

$$\mathbf{x}^T L_G \mathbf{x} = \sum_{(u,v)\in E} w(u,v)(\mathbf{x}(u) - \mathbf{x}(v))^2 \tag{2.10}$$

This expression measures the smoothness of $\mathbf{x}$. To understand the structure of L_G consider a graph with only a single edge (u, v). We can rewrite

$$\mathbf{x}(u) - \mathbf{x}(v) = (\mathbf{e}_u - \mathbf{e}_v)^T \mathbf{x} \tag{2.11}$$

and subsequently

$$(\mathbf{x}(u) - \mathbf{x}(v))^2 = ((\mathbf{e}_u - \mathbf{e}_v)^T \mathbf{x})^2 = \mathbf{x}^T (\mathbf{e}_u - \mathbf{e}_v)(\mathbf{e}_u - \mathbf{e}_v)^T \mathbf{x}. \tag{2.12}$$

Therefore the Laplacian of a graph containing only a single edge e is given by:

$$L_e = w(e)(\mathbf{e}_u - \mathbf{e}_v)(\mathbf{e}_u - \mathbf{e}_v)^T \tag{2.13}$$

We then can obtain the Laplacian matrix for any graph G by simply summing over all edges:

$$L_G = \sum_{e\in E} L_e = \sum_{e\in E} w(e)(\mathbf{e}_u - \mathbf{e}_v)(\mathbf{e}_u - \mathbf{e}_v)^T \tag{2.14}$$

One can show that this is equal to

$$L_G = D_G - A_G \tag{2.15}$$

where D_G is the diagonal degree matrix and A_G is the adjacency matrix of the graph [Spi12]. Based on this, we can see that the Laplacian matrix has the following properties:

(1) $L \in \mathbb{R}^{n\times n}$ is symmetric

(2) $L_{i,j} \leq 0$ for all $i \neq j$

(3) all rows and columns of L sum up to 0: $\forall i \sum_{k=1}^{|V|} L_{i,k} = \sum_{k=1}^{|V|} L_{k,i} = 0$

Conversely, any matrix satisfying these properties is a Laplacian matrix:

Lemma 2.3.1. *Any real symmetric matrix $L \in \mathbb{R}^{n\times n}$ that satisfies properties (1)–(3) above is the Laplacian matrix of a weighted undirected graph.*

Proof. Given a matrix $L \in \mathbb{R}^{n\times n}$, satisfying the properties above. Define $A \in \mathbb{R}^{n\times n}$ by

$$A(i,j) = \begin{cases} -L(i,j) & \text{if} \quad i \neq j, \\ 0 & \text{if} \quad i = i. \end{cases}$$

Because L is symmetric and has non-positive off-diagonal entries, A is symmetric with non-negative entries, and thus can be interpreted as the weighted adjacency matrix of an undirected graph $G = (V, E, w)$. Using property (3), we have

$$D_G(i) = \sum_j A(i,j) = \sum_{i \neq j} -L(i,j) = L(i,i).$$

It follows that $L = D_G - A_G$, which matches the definition of the Laplacian matrix in Equation 2.15. □

We may not always work directly with Laplacian matrices, but we will also encounter matrices with similar properties. Before proceeding, we introduce some relevant terminology.

Z-Matrix: A square matrix M such that all off diagonal entries are non-positive, $M(i,j) \leq 0$ for all $i \neq j$

Symmetric diagonally dominant (SDD) matrix: A symmetric matrix M satisfying $M(i,i) \geq \sum_{j \neq i} |M(i,j)|$

Directed Laplacian Matrices: Directed Laplacians $\mathcal{L}$ are an extension for directed graphs. In a directed graph, the neighborhood relation is not symmetric and a node's in-degree may differ from its out-degree. We define the directed Laplacian using the out-degree:

$$\mathcal{L} = D_{out} - A^T$$

Since A is not symmetric and a node can have different in and out-degree, the matrix $\mathcal{L} \in \mathbb{R}^{n \times n}$ has weaker properties:

(1) $\mathcal{L}$ is a *Z-Matrix*

(2) all columns of $\mathcal{L}$ sum up to 0: $\forall i \sum_{k=0}^{|V|} \mathcal{L}_{i,k} = 0$

Similar to the undirected case, any matrix that satisfies these properties is a directed Laplacian matrix.

2.4. Schur Complement

The *Schur complement* is an important matrix construction that plays a key role in our sparsification when transforming matrices associated with graphs.

Definition 2.4.1. *Let* $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ *be a square matrix partitioned into four blocks, where* A *and* D *are square submatrices. If* A *is invertible, the Schur complement of* M *on* D *is defined as:*

$$SC_D(M) = D - CA^{-1}B$$

Given a square matrix M of dimension n, we can define its block structure by partitioning the index set into two subsets, $S = \{1, 2, ..., k\}$ and $B = \{k+1, k+2, ...n\}$. For example, notice that the block A corresponds to $M(1, k : 1, k)$, and the block B to $M(1 : k, k+1 : n)$. Thus, it is often more natural to express the block structure using subset notation as:

$$M = \begin{bmatrix} M_{S,S} & M_{S,B} \\ M_{B,S} & M_{B,B} \end{bmatrix}.$$

We also adopt the shorthand notation $SC_B(M) := SC_{M_{B,B}}(M)$ in these cases. This convention is particularly useful when working with Schur complements of Laplacian matrices L_G, where the subsets S and B correspond to a partition of the vertex set of a graph G. In such cases, we may slightly abuse notation and write, for example, $L(u, v) = L_{B,B}(u, v)$ for two nodes $u, v \in B$, to refer to entries of the block $L_{B,B}$ using the original indexing of L.

The Schur complement naturally arises when eliminating entries below the block diagonal in a matrix. This can be observed by multiplying a block matrix M by

$$\begin{bmatrix} I & 0 \\ -CA^{-1} & I \end{bmatrix},$$

which yields

$$\begin{bmatrix} I & 0 \\ -CA^{-1} & I \end{bmatrix} \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} A & B \\ 0 & D - CA^{-1}B \end{bmatrix}$$

This transformation eliminates the block below the diagonal while modifying the bottom-right block, which we recognize as the Schur complement $SC_D(M)$. This connection explains why the Schur complement is sometimes referred to as *block Gaussian elimination* and we will further explore the relationship between Schur complement and Gaussian elimination next.

Schur Complement and Gaussian Elimination. Gaussian elimination is a method for transforming a square matrix A into an upper triangular form by applying a sequence of row operations, typically eliminating one column at a time. Consider two rows of a matrix:

$$\mathbf{r}_1 = [r_{11}, r_{12}, ..., r_{1n}] \tag{2.16}$$

$$\mathbf{r}_2 = [r_{21}, r_{22}, ..., r_{2n}] \tag{2.17}$$

To eliminate the first entry of $\mathbf{r}_2$, we subtract a scaled multiple of $\mathbf{r}_1$ (a *row operation*), assuming $r_{11} \neq 0$:

$$\mathbf{r}_2' = \mathbf{r}_2 - \frac{r_{21}}{r_{11}}\mathbf{r}_1 = [0, r_{22} - \frac{r_{21}}{r_{11}}r_{12}, ..., r_{2n} - \frac{r_{21}}{r_{11}}r_{1n}] \tag{2.18}$$

This operation eliminates the first entry in $\mathbf{r}_2$. The same process can be applied to all other rows, eliminating the first entry in every row except in $\mathbf{r}_1$. We then repeat this process using $\mathbf{r}_2$ to eliminate the second entry. Continuing this process results in a

triangular matrix. Using Gaussian elimination to eliminate the first n entries transforms a square block matrix $M \in \mathbb{R}^{(n+m)\times(n+m)}$, which contains a square block $A \in \mathbb{R}^{n\times n}$:

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad \xrightarrow{\text{Gaussian elimination}} \quad M' = \begin{bmatrix} A' & B' \\ 0 & D + X \end{bmatrix}$$

X represents the accumulated changes applied to the block D due to row operations (also note that A' is an upper triangular matrix). The resulting matrix M' resembles the Schur complement in that it also eliminates block C. The key question is how similar the lower-right block is in both approaches. In fact, we will show that they are identical if A is invertible by proving that $X = CA^{-1}B$. To do so, we closely follow the proof by Gantmacher [GH59]. To establish this result, we first introduce the concept of the rank of a matrix.

Definition 2.4.2. *The rank of a matrix M is the number of linearly independent rows in M.*

We will denote the rank of a matrix M by $\operatorname{rank}(M)$.

Lemma 2.4.1. *Let M be a matrix, and let M' be obtained from M by applying a sequence of row operations. Then,*

$$\mathit{rank}(M) = \mathit{rank}(M').$$

This follows immediately from the definition of the rank and the fact that row operations do not change the number of linearly independent rows. Now, we proceed to prove $X = CA^{-1}B$.

Proof. Since A is invertible, it must be full rank and $\operatorname{rank}(A) = n$. When applying partial Gaussian elimination, the values in X depend only on the first n rows and columns of M, and thus only on the blocks A, B and C, but not on D. Therefore, without loss of generality, we assume $D = CA^{-1}B$. Given the matrix

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} A & B \\ C & CA^{-1}B \end{bmatrix}, \tag{2.19}$$

we notice that $\operatorname{rank}(M) = \operatorname{rank}(A)$. This becomes clear by subtracting the first n rows of M scaled by CA^{-1} from the lower blocks C and D:

$$\begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix} \tag{2.20}$$

Since the rank is determined by the number of linearly independent rows, the rank of the matrix is at most n, and since $\operatorname{rank}(A) = n$, it is exactly n. By Lemma 2.4.1, this is also the rank of M.

Moreover, since A' is triangular, we have $\operatorname{rank}(A') = n$. As Gaussian elimination transforms M into M' through row operations, it follows again from Lemma 2.4.1 that $\operatorname{rank}(M) = \operatorname{rank}(M') = n$. These equalities can only hold if $D + X = 0$, which implies that $X = D = CA^{-1}B$. □

This allows us to interpret the Schur complement as a result of repeated column elimination. Another key observation is that the result is independent of the order of the first n eliminations, as we imposed no constraints on their sequence. Since we focus on matrices associated with graphs, where each row and column corresponds to a node, we will refer to column elimination as *vertex elimination*. Next, we examine two interesting properties of the Schur complement.

Properties of the Schur complement. First, we show that the Schur complement preserves solutions when solving a linear system of equations. This property allows us to reduce the size of the system without altering its solutions.

Lemma 2.4.2. *Let* $M = \begin{bmatrix} M_{S,S} & M_{S,B} \\ M_{B,S} & M_{B,B} \end{bmatrix} \in \mathbb{R}^{n \times n}$, *and let* $\mathbf{b} = \begin{bmatrix} \mathbf{0} \\ \mathbf{b}_B \end{bmatrix} \in \mathbb{R}^n$. *If the Schur complement* $SC_B(M)$ *exists, then for the solution* $\mathbf{x} = \begin{bmatrix} \mathbf{x}_S \\ \mathbf{x}_B \end{bmatrix}$ *to* $M\mathbf{x} = \mathbf{b}$, *the vector* $\mathbf{x}_B$ *satisfies* $SC_B(M)\mathbf{x}_B = \mathbf{b}_B$.

Proof. Let

$$M\mathbf{x} = \begin{bmatrix} M_{S,S} & M_{S,B} \\ M_{B,S} & M_{B,B} \end{bmatrix} \begin{bmatrix} \mathbf{x_S} \\ \mathbf{x_B} \end{bmatrix} = \begin{bmatrix} M_{S,S}\mathbf{x_s} + M_{S,B}\mathbf{x_B} \\ M_{B,S}\mathbf{x_S} + M_{B,B}\mathbf{x_B} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{b_B} \end{bmatrix}. \tag{2.21}$$

From the first row of (2.21) we know:

$$M_{S,S}\mathbf{x_s} + M_{S,B}\mathbf{x_B} = \mathbf{0} \tag{2.22}$$

$$M_{B,S}M_{S,S}^{-1}(M_{S,S}\mathbf{x_S} + M_{S,B}\mathbf{x_B}) = \mathbf{0} \tag{2.23}$$

$$M_{B,S}\mathbf{x_S} + M_{B,S}M_{S,S}^{-1}M_{S,B}\mathbf{x_B} = \mathbf{0} \tag{2.24}$$

Subtracting (2.24) from the second row of (2.21) shows:

$$M_{B,S}\mathbf{x_S} + M_{B,B}\mathbf{x_B} - M_{B,S}\mathbf{x_S} - M_{B,S}M_{S,S}^{-1}M_{S,B}\mathbf{x_B} = \mathbf{b_B} \tag{2.25}$$

$$(M_{B,B} - M_{B,S}M_{S,S}^{-1}M_{S,B})\mathbf{x_B} = \mathbf{b_B} \tag{2.26}$$

$$SC_B(M)\mathbf{x_B} = \mathbf{b_B} \tag{2.27}$$

We see that the Schur complement of M preserves the solution $\mathbf{x_B}$ given a $\mathbf{b_B}$. □

Another important property is that the Schur complement of a Laplacian matrix is itself a Laplacian matrix and thus has an associated graph.

Lemma 2.4.3. *Directed Laplacian matrices are closed under the Schur complement.*

Proof. Building on the observation that the Schur complement can be obtained through repeated single-vertex elimination, we show that eliminating a single vertex from L results in another Laplacian matrix. The general case then follows by induction.

Without loss of generality, we eliminate the first vertex, corresponding to the first row and column of L. We demonstrate that the resulting matrix is a directed Laplacian

by verifying that it satisfies the required properties of a directed Laplacian matrix (see Lemma 2.3.1 and its extension to directed Laplacians).

Let L be a directed Laplacian matrix of size n, $S = \{1\}$ and $B = \{2, 3, ...n\}$. The Schur complement is defined as

$$SC_B(L) = L_{B,B} - L_{B,S}L_{S,S}^{-1}L_{S,B} \tag{2.28}$$

$$= L_{B,B} - \frac{L_{B,\{1\}}L_{\{1\},B}}{L_{1,1}} \qquad \text{since } S \text{ is of size 1} \tag{2.29}$$

$L_{B,\{1\}}$ and $L_{\{1\},B}$ contain only off diagonal entries of L, all of which are non positive. Thus the entries of the matrix $L_{B,1}L_{1,B}$ are non negative. $L_{1,1}$ is on the diagonal and positive. Consequently, the above subtraction makes the existing entries of $L_{B,B}$ smaller and off diagonal entries remain non positive, thus satisfying Property (1) of directed Laplacian matrices.

It is only left to show that the columns of the resulting matrix sum up to 0 (see Property (2)). Consider the sum of column i. In the following equation we map the entries of blocks $L_{B,S}, L_{S,B}$ and $L_{S,S}$ to entries of L for clarity:

$$\sum_{j=1}^{n-1} SC_B(L)(j,i) = \sum_{j=2}^{n} L(j,i) - \frac{L(j,1)L(1,i)}{L(1,1)} \tag{2.30}$$

$$= \sum_{j=2}^{n} L(j,i) - \sum_{j=2}^{n} \frac{L(j,1)L(1,i)}{L(1,1)} \tag{2.31}$$

$$= \sum_{j=2}^{n} L(j,i) - \frac{L(1,i)}{L(1,1)} \sum_{j=2}^{n} L(j,1) \tag{2.32}$$

$$= \sum_{j=2}^{n} L(j,i) - \frac{L(1,i)}{L(1,1)}(-L(1,1)) \tag{2.33}$$

$$= \sum_{j=2}^{n} L(j,i) + L(1,i) \tag{2.34}$$

$$= \sum_{j=1}^{n} L(j,i) - L(1,i) + L(1,i) \tag{2.35}$$

$$= \sum_{j=1}^{n} L(j,i) = 0 \tag{2.36}$$

The transformation step in line 2.33 and 2.36 follows from the fact that the columns of directed Laplacians sum up to 0. Since off diagonal entries are non positive and all columns sum up to 0, the resulting matrix is again a directed Laplacian. □

In a similar way, one can show that undirected Laplacians (using their symmetry) and diagonally dominant Z-matrices (by replacing equalities with inequalities) are closed under the Schur complement.

In the proof we observed the following property, which will be useful later.

Corollary 2.4.1. *Given a Laplacian matrix* $L = \begin{bmatrix} L_{S,S} & L_{S,B} \\ L_{B,S} & L_{B,B} \end{bmatrix}$, *it holds that* $L_{B,B}(i,j) \geq SC_B(L)(i,j)$. *This means that the entries in the Schur complement are upper bounded by their corresponding entries in the input matrix* $L_{B,B}$.

An Application of the Schur complement. We will briefly outline an application of these properties in vertex sparsification, as also demonstrated, for example, in [Spi19]. Consider a resistance network $G = (V, E, w)$, a special type of graph where the edges are interpreted as resistors and nodes as points where external currents $\mathbf{i}$ can be applied. Using the Laplacian L of the graph one can solve the system $L\mathbf{v} = \mathbf{i}$ to compute the voltages $\mathbf{v}$ at each node. Let B and S refer to vertex subsets for which $B \cap S = \emptyset$ and $B \cup S = V$. By Lemma 2.4.3, the Schur complement $SC_B(L)$ is itself a Laplacian of a resistance network defined only on the nodes in B. Furthermore, by Lemma 2.4.2, if no current is applied to nodes in S, then the system defined by $SC_B(L)$ yields the same voltages on B as the original system $L\mathbf{v} = \mathbf{i}$, when the same current is applied to nodes in B. Thus, the Schur complement allows us to reduce a resistance network G to a subset of nodes B while preserving the overall electrical behavior of the network.

3. Personalized PageRank Sparsifiers

We begin this chapter by presenting an algorithm for computing exact *personalized PageRank sparsifiers* (PPR sparsifier), along with a proof of its correctness (Section 3.1). In the following Section 3.1.1, we discuss an optimization for undirected graphs.

We then analyze why computing exact sparsifiers is computationally expensive by examining Gaussian elimination from the perspective of a sequence of graph transformations (Section 3.2.1). Building on this viewpoint, we explore approximation algorithms to accelerate computation (Section 3.2.2).

Finally, while we were unable to establish formal guarantees for our proposed approximation methods, we provide an initial theoretical analysis of the limitations of PageRank approximation quality in sparse graphs (Section 3.2.3).

3.1. Exact Personalized PageRank Sparsifiers

In this section we formally define what a *PPR sparsifier* is and how one can compute such a sparsifier.

Definition 3.1.1. *Given a graph $G = (V, E, w)$, a graph $H = (V', E', w')$ is a (exact) personalized PageRank sparsifier if for any node $u, v \in V \cap V'$,*

$$\frac{\mathbf{p}_v^G(u)}{\sum_{w \in V \cap V'} \mathbf{p}_v^G(w)} = \frac{\mathbf{p}_v^H(u)}{\sum_{w \in V \cap V'} \mathbf{p}_v^H(w)},$$

where $\mathbf{p}_v^G$ and $\mathbf{p}_v^H$ are the v-personalized PageRank vectors of G and H.

This means all v-personalized PageRank values are identical for all common nodes after normalization. To give some idea why normalization is needed, consider a graph G with three nodes u, v, and w, where the PageRank values are $\mathbf{p}_G(u) = 0.4$, $\mathbf{p}_G(v) = 0.2$, and $\mathbf{p}_G(w) = 0.4$ (for some personalization). Suppose we are interested in a vertex sparsifier only on nodes u and v. In that case, the PageRank values must still sum to 1 (as PageRank is a probability distribution) and achieving identical values is not feasible. However, values such as $\mathbf{p}_H(u) = 2/3$ and $\mathbf{p}_H(v) = 1/3$ would be acceptable, as they maintain the relative importance between the nodes, specifically, u's score is twice that of v both in the input and the sparsifier.

The main result we aim to prove is the following:

Theorem 3.1.1. *Given a graph $G(V, E, w)$ and a subset $V' \subseteq V$, there is an algorithm that computes a directed personalized PageRank sparsifier $H = (V' \cup \{s\}, E', w')$, where s is an auxiliary vertex (also called the* sink *node).*

As a first step towards proving the main theorem, we describe the construction of a restricted variant called a *simple PPR sparsifier*. In this setting, the sparsifier is only required to preserve personalized PageRank values for a single, fixed personalization vector $\mathbf{r}_u$, where $\mathbf{r}_u(u) = 1$ and $\mathbf{r}_u(v) = 0$ for all $v \neq u$.

Lemma 3.1.1. *(Simple PPR Sparsifier) Given a graph $G = (V, E, w)$, a subset $V' \subseteq V$ and a fixed $v \in V \cap V'$, there exists an algorithm that computes a directed graph $H = (V', E', w')$ such that for all $u \in V \cap V'$,*

$$\frac{\mathbf{p}_v^G(u)}{\sum_{w \in V \cap V'} \mathbf{p}_v^G(w)} = \frac{\mathbf{p}_v^H(u)}{\sum_{w \in V \cap V'} \mathbf{p}_v^H(w)}$$

Proof. Recall that given a Graph $G = (V, E, w)$, PageRank is defined as the solution to the following equation:

$$\mathbf{p} = (\alpha \mathbf{r}\mathbf{1}^T + (1-\alpha) A_G^T D_G^{-1})\mathbf{p}, \tag{3.1}$$

which can be rewritten as the linear system:

$$\mathbf{0} = (I - (\alpha \mathbf{r}\mathbf{1}^T + (1-\alpha) A_G^T D_G^{-1}))\mathbf{p}. \tag{3.2}$$

In Section 2.2 we discussed that $\alpha \mathbf{r}\mathbf{1}^T + (1-\alpha) A_G^T D_G^{-1}$ corresponds to the adjacency matrix of the so-called PageRank graph G_{PageRank}, whose columns sum to one. Subtracting this matrix from the identity therefore yields the Laplacian of G_{PageRank}:

$$\mathbf{0} = (I - A_{G_{PageRank}})\mathbf{p} = L_{G_{PageRank}}\mathbf{p} \tag{3.3}$$

Consider the following algorithm:

Algorithm 3.1.1 PPR-Sparsify

Input: a (directed) Graph $G = (V, E, w)$, terminal node set $V' \subseteq V$, restart probability α, personalization vector $\mathbf{r}_u$

Output: adjacency matrix A_H of the (simple) personalized PageRank sparsifier $H = (V', E', w')$

1: $L_{G_{PageRank}} = I - (\alpha \mathbf{r}_u \mathbf{1}^T + (1-\alpha) A_G^T D_G^{-1})$
2: compute $SC_{V'}(L_{G_{PageRank}})$
3: Split $SC_{V'}(L_{G_{PageRank}}) = I - \alpha \mathbf{r}_u \mathbf{1}^T - (1-\alpha) A_H^T$
4: return A_H

Claim. The returned matrix A_H defines a valid adjacency matrix, whose rows sum to one, and the corresponding graph H is a simple PPR sparsifier.

Non-negativity. We begin by showing that the Schur complement splitting in Line 3 of Algorithm 3.1.1 yields a valid adjacency matrix A_H, i.e., a matrix with non-negative entries. Let $v, w \in V'$ with $v \neq w$. By definition,

$$L_{G_{PageRank}}(v, w) = -(\alpha \mathbf{r}_u \mathbf{1}^T + (1-\alpha) A_G^T D_G^{-1})(v, w). \tag{3.4}$$

According to Corollary 2.4.1, the entries of the Schur complement cannot increase relative to the corresponding entries in the original Laplacian:

$$\begin{aligned} L_{G_{PageRank}}(v,w) &\geq SC_{V'}(L_{G_{PageRank}})(v,w) \\ -(\alpha \mathbf{r}_u \mathbf{1}^T + (1-\alpha)A_G^T D_G^{-1})(v,w) &\geq -(\alpha \mathbf{r}_u \mathbf{1}^T + (1-\alpha)A_H)(v,w) \\ A_G^T D_G^{-1}(v,w) &\leq A_H(v,w) \\ 0 &\leq A_H(v,w) \end{aligned}$$

Row sums. Next, we show that the rows of A_H sum to one. Since $L_{G_{PageRank}}$ is a directed Laplacian, Lemma 2.4.3 implies, that $SC_{V'}(L_{G_{PageRank}})$ is also a directed Laplacian. By definition it can be expressed as:

$$SC_{V'}(L_{G_{PageRank}}) = L_{SC} = D_{SC} - A_{SC}^T, \tag{3.5}$$

where D_{SC} is a diagonal degree matrix and A_{SC} is the adjacency matrix of some graph. Since $L_{G_{PageRank}}(i,i) = 1$ and Corollary 2.4.1 ensures that diagonal entries do not increase in the Schur complement, it follows that $0 \leq D_{SC}(i,i) \leq 1$. Thus we can write $D_{SC}(i,i) = 1 - x_i$ for some $x_i \geq 0$. In matrix notation we obtain:

$$SC_{V'}(L_{G_{PageRank}}) = D_{SC} - A_{SC}^T = I - (A_{SC}^T + X) = I - (\alpha \mathbf{r}\mathbf{1}^T + (1-\alpha)A_H^T). \tag{3.6}$$

$A_{SC}^T + X$ is also an adjacency matrix but its columns sum up to one. The diagonal entries added by the diagonal matrix X correspond to self loops in the graph. Noting that $A_{SC}^T + X = \alpha \mathbf{r}\mathbf{1}^T + (1-\alpha)A_H^T$ (see line 3 of Algorithm 3.1.1), it follows that the columns of A_H^T must also sum to 1.

Preserving PPR. Finally, computing the PageRank vector $\mathbf{p}_u^H$ in H using the personalization vector $\mathbf{r}_u$ gives:

$$\begin{aligned} (I - \alpha \mathbf{r_u}\mathbf{1}^T - (1-\alpha)A_H^T D_H^{-1})\mathbf{p} &= \mathbf{0} \\ (I - \alpha \mathbf{r_u}\mathbf{1}^T - (1-\alpha)A_H^T I^{-1})\mathbf{p} &= \mathbf{0} \\ (I - \alpha \mathbf{r_u}\mathbf{1}^T - (1-\alpha)A_H^T)\mathbf{p} &= \mathbf{0} \\ SC_{V'}(L_{G_{PageRank}})\mathbf{p} &= \mathbf{0} \end{aligned}$$

By Lemma 2.4.2, the Schur complement preserves the solution to the system $L_{G_{PageRank}}\mathbf{p} = \mathbf{0}$ associated with the input graph G (at the terminal nodes). Therefore, H is a simple PPR sparsifier. □

Next, we examine how the algorithm can be extended by introducing an additional node in the sparsifier to preserve PageRank values for all u-personalization vectors. This construction leads us to the proof of Theorem 3.1.1.

Proof. Let $G = (V, E, w)$ be a graph with normalized adjacency matrix $A^T D^{-1}$. We construct a new graph $G' = (V + \{s\}, E', w')$ by adding a new node s and define its adjacency matrix based on a given restart probability α and personalization vector $\mathbf{r}$ as follows:

$$A_{G'}^T = \begin{bmatrix} 0 & \alpha \mathbf{1}^T \\ \mathbf{r} & (1-\alpha) A^T D^{-1} \end{bmatrix} \tag{3.7}$$

The first row and column represent the edges adjacent to the new node s. Figure 3.1 illustrates this construction using our earlier toy example. When eliminating s from the Laplacian $L_{G'}$, we obtain:

$$SC_{\{s\}}(L_{G'}) = (I - (1-\alpha) A^T D^{-1}) - \alpha \mathbf{r} \mathbf{1}^T = L_{G_{PageRank}} \tag{3.8}$$

This shows that eliminating s yields the PageRank Laplacian of G. By Lemma 2.4.2, the linear systems defined by $L_{G_{PageRank}}$ and $L_{G'}$ admit the same solution. Hence, we can substitute G' for G in Algorithm 3.1.1 to construct a simple PageRank sparsifier. This modification of the PageRank problem also appears in [LM04], and such a matrix augmentation is a common optimization in solving linear systems where the matrix is expressed as the sum of a sparse and a low-rank component [BV04], precisely the type of structure that appears in the PageRank equation (e.g., see Equation 3.3).

However, this construction is still dependent on the personalization vector $\mathbf{r}$, which we would like to avoid. Our goal is to find a representation that works for all $\mathbf{r}_u$, where $u \in V'$ is a terminal node. Since we restrict our sparsifiers to such terminal personalization vectors, we have $\mathbf{r}_u(v) = 0$ for all non terminal $v \in V \setminus V'$ and thus $L_{G'}(v, s) = 0$.

Recall that the Schur complement can be computed through Gaussian elimination, specifically using operations of row scaling and row subtraction. Given that $L_{G'}(v, s) = 0$ for all $v \in V \setminus V'$, eliminating any non-terminal node v will not affect the column corresponding to node s. Therefore, for all $u \in V'$, the entry in the Schur complement remains unchanged:

$$SC_{\{v\}}(L_{G'})(u, s) = L(u, s) = \mathbf{r}(u) \tag{3.9}$$

Now consider what happens if we remove all outgoing edges from s, making it a dangling node. For PageRank computations, it is crucial that the graph is strongly connected and the transition matrix is well-defined. To handle dangling nodes, it is common to apply a preprocessing step that adds edges from s to all nodes u, with weights $\mathbf{r}(u)$. Since we have just shown that $SC_{\{v\}}(L_{G'})(u, s) = \mathbf{r}(u)$, this preprocessing step effectively reconstructs the necessary restart edges for any $\mathbf{r}$.

Thus, by exploiting this behavior of the preprocessing step, we obtain a sparsifier that becomes independent of the specific personalization vector $\mathbf{r}$: the required information is implicitly reconstructed during computation. This completes the proof of Theorem 3.1.1. □

One difference between our approach and that of the authors in [VCG11] is how we handle the outgoing edges of the node s. In their method, all outgoing edges of s are removed and replaced with a single self-loop. In contrast, as we have just shown, it is possible to remove all outgoing edges without adding a self-loop. Our approach is

consistent with the common preprocessing step in PageRank computations, where edges are added to dangling nodes (nodes with no outgoing edges) based on the personalization vector. Ultimately, however, both approaches rely on the handling of dangling nodes.

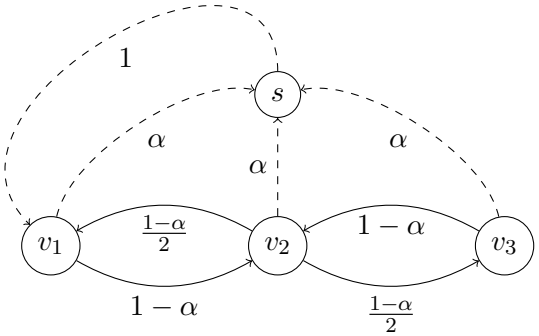

Figure 3.1.: Modified Pagerank Graph $G'_{PageRank}$ with additional node s for graph G from 2.1a

3.1.1. Simplifying Computation for Undirected Input Graphs

Now, that we know how to compute a personalized PageRank sparsifier, we would like to achieve this efficiently. For instance we would prefer to compute the Schur complement of an undirected Laplacian if the input graph is undirected, allowing us to use the symmetry and other properties of undirected Laplacians. However, the additional restart edges and normalizing A^T with D^{-1} create a directed graph. We will look at how we can avoid both these obstacles.

Lemma 3.1.2. *If the input to Algorithm 3.1.1 is an undirected graph $G = (V, E, w)$, then the algorithm can be modified such that it only requires computing the Schur complement of a SDD Z-Matrix.*

Recall that SDD Z-matrices are essentially undirected Laplacians with some slack on the diagonal. From the perspective of our algorithm, there is no difference whether we compute the Schur complement of a Laplacian or an SDD Z-matrix. However, using SDD Z-matrices allows for simpler notation in the proof. We break the argument into two steps:

Step 1: Omit restart edges. Given a graph G, recall that our algorithm works with the Laplacian matrix of the augmented graph G' with the additional node s (as shown in Equation 3.7). With the introduction of the new node s, the number of restart edges is reduced to those adjacent to s. Specifically, s has incoming edges corresponding to $\alpha\mathbf{1}$, and outgoing edges defined by the restart distribution $\mathbf{r}$, which are directed. Therefore, we aim to exclude these edges from the computation of the Schur complement.

We show that we can simply remove the new row and column associated with s and only compute $SC_B(I-(1-\alpha)A^TD^{-1})$. Specifically, we show two things: (1) for terminal nodes $u, v \neq s$, $SC_B(L_{G'})(u,v) = SC_B(I-(1-\alpha)A^TD^{-1})(u,v)$, meaning the values corresponding to u and v remain identical, and (2) how we can infer the values of the row and column associated with s afterward.

Let us begin with the first point. The Schur complement is equivalent to Gaussian elimination, which involves applying row operations. A row operation modifies a row by adding a scaled multiple of a non-terminal row. Since s is a terminal node (i.e., it remains in the sparsifier), the omission of its row has no effect on the other entries. Furthermore, this implies that the values in the column corresponding to s can only affect other values in the same column. Therefore, the removal of both the row and column associated with s does not affect the other entries in the Schur complement.

Now, regarding the second point: we previously observed that the column corresponding to s does not change during elimination (see Proof 3.1), so we can naturally recover it. While the row, initially equal to $\alpha\mathbf{1}$, does change under the Schur complement, we can also recover these values. Since the input is a directed Laplacian matrix, the Schur complement remains a directed Laplacian (by Lemma 2.4.3). Therefore we can use the fact that a column must sum to 0 and the following identity holds:

$$\forall v \in V : SC_B(L_G)(s,v) = -\sum_{\substack{u \in V \\ u \neq s}} SC_B(L_G)(u,v). \tag{3.10}$$

Combining these two observations, that the row and column associated with s do not affect the other entries, and that their values can be inferred afterward, we simplify the computation. Instead of including the new row and column in the Schur complement computation, we omit them and reconstruct their values afterward. The only caveat is that we are now computing the Schur complement of a symmetric diagonally dominant (SDD) Z-matrix, rather than a Laplacian matrix.

Alternatively, one can show that the outgoing edges from s can be replaced in a way that makes the graph symmetric and then removed afterward. These outgoing edges have no effect on other entries in the Schur complement.

Step 2: Make non-restart edges symmetric. To achieve this, we rely on the following lemma:

Lemma 3.1.3. *Let* $M = \begin{bmatrix} M_{S,S} & M_{S,B} \\ M_{B,S} & M_{B,B} \end{bmatrix}$ *be a matrix and* $D = \begin{bmatrix} D_{S,S} & 0 \\ 0 & D_{B,B} \end{bmatrix}$ *be a diagonal matrix. If the Schur complement* $SC_B(M)$ *exists, then* $SC_B(MD) = SC_B(M)D_B$.

Proof. We begin by expressing the matrix product:

$$\begin{aligned} MD &= \begin{bmatrix} M_{S,S} & M_{S,B} \\ M_{B,S} & M_{B,B} \end{bmatrix} \begin{bmatrix} D_{S,S} & 0 \\ 0 & D_{B,B} \end{bmatrix} \\ &= \begin{bmatrix} M_{S,S}D_{S,S} & M_{S,B}D_{B,B} \\ M_{B,S}D_{S,S} & M_{B,B}D_{B,B} \end{bmatrix} \end{aligned}$$

Now, applying the definition of the Schur complement:

$$\begin{aligned} SC_B(MD) &= M_{B,B}D_{B,B} - M_{B,S}D_{S,S}(M_{S,S}D_{S,S})^{-1}M_{S,B}D_{B,B} \\ &= M_{B,B}D_{B,B} - M_{B,S}D_{S,S}D_{S,S}^{-1}M_{S,S}^{-1}M_{S,B}D_{B,B} \\ &= M_{B,B}D_{B,B} - M_{B,S}M_{S,S}^{-1}M_{S,B}D_{B,B} \\ &= (M_{B,B} - M_{B,S}M_{S,S}^{-1}M_{S,B})D_{B,B} \\ &= SC_B(M)D_{B,B} \end{aligned}$$

□

This lemma allows us to defer the normalization step (i.e., multiplication with D^{-1}) until after computing the Schur complement. As a result, we can operate directly on the input graph without immediately introducing asymmetry to the non-restart edges.

By combining both steps, we can now prove the correctness of Lemma 3.1.2.

Proof. In **Step 1**, we observed that the restart edges can be omitted during the computation of the Schur complement. As a result, we are interested in the Schur complement of the matrix $I - (1-\alpha)A^TD^{-1}$. Using Lemma 3.1.3, we can rewrite this as:

$$\begin{aligned} &SC_B(I - (1-\alpha)A^TD^{-1}) \\ =&SC_B((I - (1-\alpha)A^TD^{-1})DD^{-1}) \\ =&SC_B((D - (1-\alpha)A^T)D^{-1}) \\ =&SC_B(D - (1-\alpha)A^T)D^{-1} \end{aligned}$$

If the input graph is undirected, then A is symmetric, and $D - (1-\alpha)A^T$ is an SDD Z-matrix. In this case, we only need to compute the Schur complement of this matrix. After computing the Schur complement, we can obtain the sparsifier H in Algorithm 3.1.1 by normalizing with D, adding the new node s, and inferring the edges as described in **Step 1**. From there, the algorithm proceeds as before. □

Concluding we can say that we have found a way to compute personalized PageRank sparsifiers of undirected and directed input graphs by reducing the problem to applying a known matrix augmentation and computing the Schur complement of a Laplacian matrix. Additionally, in the case that the input is undirected, only the Schur complement of a symmetric matrix is required.

3.2. Approximate PageRank Sparsifiers

In [VCG11] the authors observed that even when the input graph is sparse, the resulting PageRank sparsifier tends to be very dense. To address this, the authors propose a simple threshold-based rounding technique: after computing the exact sparsifier, they remove all edges with weight below a chosen threshold. This post-processing step reduces the density of the output graph. However, this approach has a major limitation. It requires

computing the exact sparsifier before any approximation can be applied. However, high density can already pose challenges during the computation of exact sparsifiers. As the algorithm transforms a sparse input graph into a dense output graph, the number of edges may grow significantly and even early in the process. This increase in density can be especially problematic when the complexity of intermediate or subsequent steps depends on the number of edges. Therefore, it would be beneficial to incorporate sparsification during the construction phase, reducing computational overhead from the beginning. This kind of approximation will be discussed here. While both approaches yield approximate sparsifiers, they serve slightly different goals. The first aims to find a sparse vertex sparsifier, whereas ours aims to reduce the cost of computing the sparsifier itself, even if the result is still dense.

In Section 3.2.1, we explore why the Schur complement required for constructing PPR sparsifiers is expensive to compute. In Section 3.2.2, we show how to approximate the Schur complement and integrate it into our pipeline to obtain approximate PPR sparsifiers.

3.2.1. Schur Complement in terms of Graphs

In Algorithm 3.1.1, we observe that the density of the output graph must arise as a result of computing the Schur complement. In Chapter 2, we noted that computing the Schur complement for a set of nodes S can be achieved by successively eliminating individual vertices. There, we approached vertex elimination from the perspective of modifying a matrix through row operations. In this section, we take a different view and examine how the graph associated with the Laplacian matrix changes under vertex elimination. Our goal is to prove the correctness of Algorithm 3.2.2, following the proof strategy outlined in [KS16].

Let $G(V, E, w)$ be an undirected graph and we define $\mathbf{b}_{(u,v)} = \mathbf{e}_u - \mathbf{e}_v$. Recall that the Laplacian matrix L_G can be expressed as a sum of edge Laplacians:

$$L_G = \sum_{e \in E} L_e = \sum_{e \in E} w(e) \mathbf{b}_e \mathbf{b}_e^T. \tag{3.11}$$

For a given vertex $v \in V$, we define the star Laplacian as:

$$L_{STAR(v)} = \sum_{u \in N(v)} w(u, v) \mathbf{b}_{(u,v)} \mathbf{b}_{(u,v)}^T. \tag{3.12}$$

This Laplacian corresponds to the star graph centered at v. Furthermore, let $\begin{bmatrix} d_v \\ -\mathbf{a_v} \end{bmatrix}$ denote the column of v in L. Then, we can express the star Laplacian as:

$$L_{STAR(v)} = \begin{bmatrix} d_v & -\mathbf{a_v}^T \\ -\mathbf{a_v} & \mathbf{diag}(\mathbf{a_v}) \end{bmatrix}. \tag{3.13}$$

When removing a single vertex v, the Schur complement is given by:

$$SC_{V/\{v\}}(L) = L - \frac{L(v,:)L_v(:,v)}{L(v,v)} = (L - L_{STAR(v)}) + (L_{STAR(v)} - \frac{L(v,:)L_v(:,v)}{L_{v,v}}) \tag{3.14}$$

This formulation of the Schur complement differs slightly from the one used initially. To maintain full consistency, we would typically exclude the first row and column, as the Schur complement reduces the matrix size by one when removing a vertex. However, for the sake of notation and clarity, we will retain the full matrix representation, as our focus is on understanding how the input matrix L transforms into $SC(L)$.

The first term, $L - L_{STAR(v)}$ represents the input graph without the edges of the star around node v. The second term

$$L_{STAR(v)} - \frac{L(v,:)L_v(:,v)}{L_{v,v}} = \begin{bmatrix} d_v & -\mathbf{a_v}^T \\ -\mathbf{a_v} & \mathbf{diag(a_v)} \end{bmatrix} - \frac{1}{d_v}\begin{bmatrix} d_v \\ -\mathbf{a_v} \end{bmatrix}\begin{bmatrix} d_v \\ -\mathbf{a_v} \end{bmatrix}^T \tag{3.15}$$

$$= \begin{bmatrix} 0 & \mathbf{0} \\ \mathbf{0} & \mathbf{diag(a_v)} - \mathbf{a_v}\mathbf{a_v}^T/d_v \end{bmatrix}. \tag{3.16}$$

We can immediately observe that the off-diagonal entries are non-positive and the matrix is symmetric. Additionally, we can confirm that the row and column sums are zero:

$$\sum_j L(i,j) = 0 + \mathbf{a}(i) - \sum_j \frac{\mathbf{a}(i)\mathbf{a}(j)}{d_v} = \mathbf{a}(i) - \frac{\mathbf{a}(i)}{d_v}\sum_j \mathbf{a}(j) = \mathbf{a}(i) - \frac{\mathbf{a}(i)}{d_v}d_v = 0 \tag{3.17}$$

Thus, the matrix is by Lemma 2.3.1 also a Laplacian, which we can express as a sum of edge Laplacians by looking at the off-diagonal entries:

$$\frac{1}{2}\sum_i\sum_j \frac{\mathbf{a}(i)\mathbf{a}(j)}{d_v}\mathbf{b}_{ij}\mathbf{b}_{ij}^T = \frac{1}{2}\sum_{u\in N(v)}\sum_{w\in N(v)} \frac{w(v,u)w(v,w)}{\deg_w(v)}\mathbf{b}_{uw}\mathbf{b}_{uw}^T \tag{3.18}$$

We observe that for each pair of adjacent nodes u, w, there is an edge (an entry with non-zero weight) with weight $w(v,u)w(v,w)/\deg_w(v)$. In graph theory, this is known as a (weighted) clique on the neighbors of node v. Thus, we can interpret vertex elimination as a transformation that removes the star around vertex v and adds a weighted clique between the neighbors of v. This leads to Algorithm 3.2.2 for computing the Schur complement of a Laplacian matrix.

Algorithm 3.2.2 Graph-Based Schur Complement

Input: graph $G = (V, E, w)$, node subset $S \subseteq V$
Output: Schur complement $SC_{V\setminus S}(L_G)$

1: **for** each v in S **do**
2: $\quad G \leftarrow G - Star(v) + Clique(v)$
3: **end for**
4: return L_G

The same applies to directed graphs, where the star is replaced by a directed clique. In this case, each edge in the clique has weight

$$\forall (u,v) \in E, \forall (v,w) \in E: \quad w'(v,w) = \frac{w(u,v)w(v,w)}{\deg_w^{out}(v)} \tag{3.19}$$

Since a star, representing the sparsest possible connected structure, is replaced by a clique, which has the maximum number of edges among a given set of nodes, this explains why repeated vertex elimination rapidly increases graph density and makes computation more expensive. More precisely, each elimination of a node v introduces $\frac{\deg(v)(\deg(v)-1)}{2} - \deg(v)$ additional edges. However, this perspective also suggests strategies to mitigate this growth.

3.2.2. Approximating Cliques by Sparse Graphs

We observed that during the construction of the Schur complement, the graph is transformed by repeatedly removing a star and replacing it with a clique. To reduce density, a natural approach is to replace the clique with a sparse approximation, which essentially modifies line 2 of Algorithm 3.2.2 as follows:

2: $G \leftarrow G - \mathit{Star}(v) + \mathit{ApproximateClique}(v)$

The core idea behind our approximate PPR sparsifier is to replace the exact Schur complement computation in Algorithm 3.1.1 with such a modified version of Algorithm 3.2.2. This section discusses efficient methods for constructing approximate cliques.

Since the goal is to reduce runtime, we need an efficient way to obtain good approximations of the clique. Ideally, we want to construct a sparsified clique without explicitly forming the full one. An approach, originally used in the context of *Cholesky factorization* [TB97], is clique sampling: instead of adding the entire clique, we sample only a few relevant edges. Since we replace many edges with only a few, we must appropriately reweight them to account for the missing edges. We will look at two sampling strategies and point out key features.

Sampling Strategy 1. We first examine the sampling strategy proposed by Kyng [KS16], which is presented in Algorithm 3.2.3.

Algorithm 3.2.3 Random Clique Sampling

Input: undirected Graph $G = (V, E, w)$, node v
Output: Laplacian $\hat{L}_{clique}$ of approximated clique when eliminating node v

1: $E_{clique} \leftarrow \emptyset$
2: **for** $i \leftarrow 1$ to $\deg(v)$ **do**
3: Sample edge $e_1 = (u_1, v)$ from $E(v) = \{(u, v) \in E\}$ with probability $\frac{w(e_1)}{\deg_w(v)}$
4: Sample edge $e_2 = (v, u_2)$ uniformly at random from $E(v)$
5: **if** $u_1 \neq u_2$ **then**
6: $w' \leftarrow \frac{w(e_1)w(e_2)}{w(e_1)+w(e_2)}$
7: $E_{clique} \leftarrow E_{clique} \cup \{(u_1, u_2, w')\}$
8: **end if**
9: **end for**
10: **return** $\hat{L}_{clique} = \sum_{e \in E_{clique}} L_e$

We can verify that the weights are chosen such that, in expectation, this sampling procedure returns the desired clique:

$$\mathbb{E}[\hat{L}_{clique}] \tag{3.20}$$
$$=\mathbb{E}[\sum_{e\in E_{clique}} L_e] \tag{3.21}$$
$$=\deg(v)\mathbb{E}[L_e] \tag{3.22}$$
$$=\deg(v)\mathbb{E}[w(e)\mathbf{b_e}\mathbf{b_e}^T] \tag{3.23}$$
$$=\deg(v)\sum_{(u,v),(w,v)\in E,w\neq u} \frac{w(u,v)}{\deg_w(v)}\frac{1}{\deg(v)}\frac{w(u,v)w(w,v)}{w(u,v)+w(w,v)}\mathbf{b_{(u,w)}}\mathbf{b_{(u,w)}}^T \tag{3.24}$$
$$=\sum_{(u,v),(w,v)\in E,(u,v)<(w,v)} (\frac{w(u,v)}{\deg_w(v)}+\frac{w(w,v)}{\deg_w(v)})\frac{w(u,v)w(w,v)}{w(u,v)+w(w,v)}\mathbf{b_{(u,w)}}\mathbf{b_{(u,w)}}^T \tag{3.25}$$
$$=\sum_{(u,v),(w,v)\in E,(u,v)<(w,v)} \frac{w(u,v)w(w,v)}{\deg_w(v)}\mathbf{b_{(u,w)}}\mathbf{b_{(u,w)}}^T \tag{3.26}$$
$$=L_{clique} \tag{3.27}$$

Thus, the sampling procedure is an unbiased estimator of the clique. Using this fact, one can show that an elimination procedure utilizing this sampling strategy will, in expectation, produce the same Cholesky factorization and Schur complement as the exact elimination process.

Each iteration retains at most $\deg(v)$ edges, rather than the $O(\deg(v)^2)$ edges present in a full clique, ensuring that the number of edges in the elimination process does not increase. This enables us to bound both the work and the number of edges during the elimination process. Let S denote the set of non-terminal vertices, and let E serve as an upper bound for the number of edges incident to a node in S. If in each iteration we select a vertex with at most average degree, the total work is given by:

$$\sum_{i=0}^{|S|-1}\frac{|E|}{|S|-i} = |E|\sum_{i=0}^{|S|-1}\frac{1}{|S|-i} = O(|E|\log(|S|)), \tag{3.28}$$

which yields a near-linear running time.

While expectation provides a relatively weak guarantee, one can strengthen the analysis by duplicating or splitting each edge into p multi-edges, each with weight $\frac{w(e)}{p}$. Using this approach, it can be shown that when applying partial Cholesky factorization, the resulting Laplacian is a spectral sparsifier of the Schur complement. For completeness, we restate (aligned to our notation) the following lemma from Kyng [Kyn17] without proof:

Lemma 3.2.1. *Suppose $G = (V, E)$ is a connected undirected multi-graph with positive edge weights $w : E \to R^+$, and associated Laplacian U, and that G has m multi-edges. Let $F \subseteq V$ be a subset of the vertices of U with $|F| = k$. Given scalars $\delta < 1/n^{100}$, $0 < \epsilon \leq 1/2$, the Laplacian matrix $\hat{S}$ returned by Algorithm 3.2.2, which utilizes clique*

sampling as described in Algorithm 3.2.3 and a uniform vertex elimination order that always eliminates a vertex with at most twice the average degree, satisfies $\hat{S} \approx_\epsilon SC_F(U)$ *(*$\hat{S}$ *is a spectral sparsifier of* $SC_F(U)$ *of quality* ϵ*), with probability at least* $1 - O(\delta)$*. The maximum number of non-zero entries in* $\hat{L}$ *and* $\hat{S}$ *and the total running time are each bounded by* $O(m \log(n)\epsilon^{-2} \log^2(1/\delta))$*.*

Sampling Strategy 2. Kyng's approach is of theoretical interest and provides formal bounds (in terms of spectral approximation). In contrast, an alternative sampling-based method proposed by Spielman et al. [GKS23] lacks theoretical guarantees but has proven competitive in practice for solving Laplacian systems. This method is based on the concept of the *elimination star*, which we will briefly introduce. The key idea is to decompose the clique into a sum of stars. Given an elimination vertex v and an arbitrary ordering π of v's neighbors, the clique can be expressed as:

$$L_{Clique(v)} = \sum_{e \in E_{Clique(v)}} L_e = \sum_{u \in N(v)} \sum_{\substack{w \in N(v), \\ \pi(w) > \pi(u)}} \frac{w(v,u)w(v,w)}{\deg_w(v)} \mathbf{b}_{uw} \mathbf{b}_{uw}^T = \sum_{u \in N(v)} L_{\text{ElimStar}(v,u)} \tag{3.29}$$

where each term in the sum corresponds to an elimination star. This decomposition and the star structure are illustrated in Figure 3.2, providing intuition for the approach. This decomposition gives rise to another clique sampling strategy, where one edge is sampled from each elimination star. For $\text{ElimStar}(v,u)$ we sample edge (u,w), where $\pi(u) < \pi(w)$, with probability:

$$p_u(w) = \frac{w(v,w)}{\sum_{x \in N(v), \pi(x) > \pi(u)} w(v,x)} \tag{3.30}$$

By setting the weights to:

$$w(u,w) = \frac{1}{p_u(w) \deg_w(v)} w(v,u)w(v,w) = \frac{w(v,u) \sum_{x \in N(v), \pi(x) > \pi(u)}}{\deg_w(v)}, \tag{3.31}$$

we can show that in expectation such a sampler will return a correct elimination star:

$$E[L_{ElimStar(v,u)}] = \sum_{w \in N(v), \pi(w) > \pi(u)} p_u(w) \frac{w(v,u)w(v,w)}{p_u(w) \deg_w(v)} \mathbf{b}_{(\mathbf{u},\mathbf{w})} \mathbf{b}_{(\mathbf{u},\mathbf{w})}{}^T \tag{3.32}$$

$$= \sum_{w \in N(v), \pi(w) > \pi(u)} \frac{w(u,v)w(v,w)}{\deg_w(v)} \mathbf{b}_{(\mathbf{u},\mathbf{w})} \mathbf{b}_{(\mathbf{u},\mathbf{w})}{}^T \tag{3.33}$$

$$= L_{ElimStar(v,u)} \tag{3.34}$$

By the linearity of expectation and Equation 3.29, it follows that this sampling strategy returns a set of edges that match those of the full clique. An advantage of this approach is that it always produces a connected graph as the returned edges form a tree on the neighbors, whereas the first sampling strategy may result in a disconnected graph. Similar to the first strategy, one can split edges and work with a multigraph; in this case, the sampled clique corresponds to a tree plus some additional edges. While this increases computational effort, it can yield better approximations [GKS23].

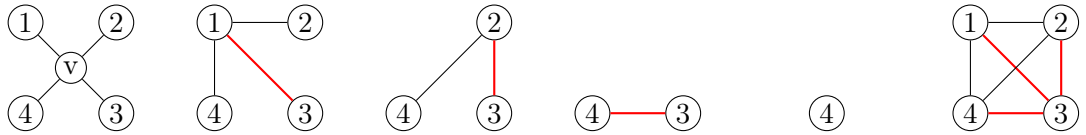

Figure 3.2.: An input star and its elimination stars, with red edges indicating the sampled ones: (a) initial input star, (b) first elimination star, (c) second elimination star, (d) third elimination star, (e) final elimination star, and (f) graph of the clique and the sampled clique.

Challenges in Using Sampling-Based Approximation Schemes. To compute an approximate personalized PageRank sparsifier, we replace the exact Schur complement computation in Algorithm 3.1.1 with an approximate variant. However, unlike the exact method, the sampling-based schemes introduce an undesired side effect that requires additional handling. Recall that after computing the Schur complement, the construction of the sparsifier in Algorithm 3.1.1 is based on a decomposition of the matrix $SC_{V'}(L_{G_{PageRank}}) = I - \alpha \mathbf{r}\mathbf{1}^T - (1-\alpha)A'^T$ or simplified $I - A^T D^{-1}$, which is equivalent to a Laplacian of a graph where each node has a weighted out-degree of exactly 1. When applying the (exact) Schur complement, the weighted degree of a vertex can only decrease. Our algorithm compensates for this decrease by adding self-loops (see equation 3.6). In terms of random walks, these self-loops account for walks that travel from a node u to the eliminated node v and return to u. However, when replacing a dense clique with a sparse approximation, we use fewer edges and therefore must reweight them. Due to the randomized nature of our approximation, it is unlikely but possible to sample a vertex with very low degree and assign it an edge with disproportionately large weight. In this case, the assumption that degrees only decrease no longer holds. It is unclear whether this issue can be avoided by modifying sampling procedures.

As a workaround, we currently omit the self-loop if its weight would exceed 1, based on the intuition that the remaining edges already absorb the corresponding probability mass. However, this does not eliminate the possibility of edge weights (and thus transition probabilities) exceeding 1. Such cases must be rounded or normalized, potentially at the cost of reduced approximation accuracy.

In summary, we have explored two very fast techniques to approximate the Schur complement in near-linear time, each of which matches the exact Schur complement in expectation, and showed how to integrate them into our PPR sparsifier algorithm. We do not have a direct bound on the quality of these sparsifiers, but we hope that preserving the Schur complement in expectation will also preserve random walks and PPR scores.

3.2.3. Limits to PageRank Approximation

Above, we explored approaches to approximating the Schur complement by replacing a clique with a sparse approximation. As the Schur complement is the key tool for constructing our PageRank sparsifier, this replacement inevitably introduces some loss in

quality. Here, we briefly outline the limitations of approximating personalized PageRank values when reducing the number of edges in a graph.

Definition 3.2.1. *(Multiplicative PageRank (Edge) Sparsifier). Given a graph $G(V, E)$ and two parameters $\gamma, \delta \geq 1$, a graph $H(V, E')$ is a multiplicative PageRank sparsifier of G with quality $(\gamma\delta)$, if for every pair of vertices $u, v \in V$, the personalized PageRank values satisfy,*

$$\frac{1}{\delta}\mathbf{p}_u^G(v) \leq \mathbf{p}_u^H(v) \leq \gamma\mathbf{p}_u^G(v).$$

We will prove a lower bound by demonstrating a sparsification limit for the unweighted complete graph. This graph is particularly attractive due to its symmetry, which simplifies PageRank computations. The key idea is as follows: in a complete graph, all nodes are exactly one hop apart. However, when sparsifying, we must reduce the number of edges, causing some nodes to become two or more hops away. The proof is based on bounding the probability mass between a node u and its one-hop neighbors versus nodes that are at least two hops away. We will start by determining personalized PageRank values for the complete graph.

Lemma 3.2.2. *Let $K_n(V, E)$ be the unweighted, complete graph with n vertices. Then for any pair of nodes $u, v \in V$, the personalized PageRank values satisfy,*

$$\mathbf{p}_u^{K_n}(v) = \begin{cases} \frac{\alpha(n-2)+1}{n-\alpha}, & \text{if } u = v, \\ \frac{1-\alpha}{n-\alpha}, & \text{if } u \neq v. \end{cases}$$

Proof. Recall that the u-personalized PageRank of a graph $G(V, E)$ can be computed by solving the linear system

$$(I - (1-\alpha)A^T D^{-1})\mathbf{p}_u = \alpha\mathbf{e}_u. \tag{3.35}$$

Since in our case $G = K_n$ is a complete graph, all nodes have the same degree $n-1$ and the equation simplifies

$$(I - \frac{1-\alpha}{n-1}A)\mathbf{p}_u = \alpha\mathbf{e}_u. \tag{3.36}$$

First we observe that for all $v \neq u$ have the same solution:

$$\mathbf{p}_u(v) - \frac{1-\alpha}{n-1}\sum_{w \neq v}\mathbf{p}_u(w) = 0 \tag{3.37}$$

$$\mathbf{p}_u(v) + \frac{1-\alpha}{n-1}\mathbf{p}_u(v) - \frac{1-\alpha}{n-1}\sum_{w}\mathbf{p}_u(w) = 0 \tag{3.38}$$

$$\mathbf{p}_u(v) + \frac{1-\alpha}{n-1}\mathbf{p}_u(v) - \frac{1-\alpha}{n-1} = 0 \tag{3.39}$$

$$\mathbf{p}_u(v) = \frac{1-\alpha}{n-\alpha} \tag{3.40}$$

Using this we obtain the following solution for u:

$$\mathbf{p}_u(u) - \frac{1-\alpha}{n-1}\sum_{v\neq u}\mathbf{p}_u(v) = \alpha \tag{3.41}$$

$$\mathbf{p}_u(u) - \frac{1-\alpha}{n-1}(1-\alpha) = \alpha \tag{3.42}$$

$$\mathbf{p}_u(u) = \alpha + \frac{(1-\alpha)^2}{n-\alpha} \tag{3.43}$$

$$\mathbf{p}_u(u) = \frac{\alpha(n-2)+1}{n-\alpha} \tag{3.44}$$

□

Next, we will bound the u-personalized PageRank value of an immediate neighbor of u and a node that is further away.

Lemma 3.2.3. *Given a Graph $G(V,E)$ with n nodes and average degree $\overline{d}$, let $u \in V$ be a vertex with $\deg(u) \leq \overline{d}$ and let $\mathbf{p}_u^G$ be its personalized PageRank vector with restart probability α. Then there exists a neighbor $v \in N(u)$ and node $w \notin N(u) \cup \{u\}$ such that*

$$\mathbf{p}_u(v) \geq \frac{1-\alpha}{\overline{d}}\mathbf{p}_u(u) \quad \text{and} \quad \mathbf{p}_u(w) \leq \frac{(1-\alpha)^2}{n-1-\overline{d}}.$$

Proof. We proceed similarly to the proof of the previous Lemma and use the PageRank equation system (3.35) to establish the bounds. The main difference here is that we are dealing with general graphs. For the vertex u, we have

$$\mathbf{p}_u(u) - ... = \alpha, \tag{3.45}$$

which immediately implies that $\mathbf{p}_u(u) \geq \alpha$. Now let $v \in N(u)$, meaning $A_G(v,u) = 1$. We then obtain:

$$-\frac{(1-\alpha)A(v,u)}{\deg(u)}\mathbf{p}_u(u) + \mathbf{p}_u(v) - ... = 0 \tag{3.46}$$

$$-\frac{(1-\alpha)}{\deg(u)}\mathbf{p}_u(u) + \mathbf{p}_u(v) - ... = 0 \tag{3.47}$$

$$\mathbf{p}_u(v) \geq \frac{(1-\alpha)}{\deg(u)}\mathbf{p}_u(u) \tag{3.48}$$

$$\mathbf{p}_u(v) \geq \frac{(1-\alpha)}{\overline{d}}\alpha, \tag{3.49}$$

which establishes the first inequality. Using this bound, we can further bound the probability mass over the set $V \setminus \{u \cup N(u)\}$:

$$1 - (\alpha + \deg(u)\frac{\alpha(1-\alpha)}{\deg(u)}) = (1-\alpha)^2 \tag{3.50}$$

Thus, there exists a two-hop or greater neighbor v such that

$$\mathbf{p}_u(v) \leq \frac{(1-\alpha)^2}{n-1-\deg(u)} \leq \frac{(1-\alpha)^2}{n-\overline{d}-1}, \tag{3.51}$$

which proves the second inequality. □

Now, we combine the previous results to prove the main lemma of this section.

Lemma 3.2.4. *Given the unweighted, complete Graph K_n on n vertices, any unweighted, sparse multiplicative sparsifier H of K_n must have quality $\Omega(n^{2/3})$.*

Proof. Let $H(V, E)$ be a sparse personalized PageRank sparsifier of the weighted complete graph K_n with quality parameters γ and δ. By Definition 3.2.1 for every $u, v \in V$ we have

$$\frac{1}{\delta}\mathbf{p}_u^{K_n}(v) \leq \mathbf{p}_u^H(v) \leq \gamma\mathbf{p}_u^{K_n}(v). \tag{3.52}$$

Recall that $\mathbf{p}_u^{K_n}(u) = \frac{\alpha(n-2)-1}{n-\alpha} \approx \frac{\alpha(n-\alpha)}{n-\alpha} = \alpha$ and $\mathbf{p}_u^{K_n}(v) = \frac{1-\alpha}{n-\alpha} \approx \frac{1-\alpha}{n}$ (see Lemma 3.2.2). Now let $u \in V$ be a low degree vertex with $\deg_H(u) \leq \overline{d}$, where $\overline{d}$ denotes the average degree in H (and is small since H is sparse). Furthermore let $v \in N_H(u)$. Using the sparsification guarantee and the estimates for K_n, we have

$$\mathbf{p}_u^H(v) \leq \gamma\mathbf{p}_u^{K_n}(v) \approx \gamma\frac{1-\alpha}{n}. \tag{3.53}$$

From Lemma 3.2.3 and sparsification guarantee 3.52 we also have

$$\mathbf{p}_u^H(v) \geq \frac{1-\beta}{\overline{d}}\mathbf{p}_u^H(u) \geq \frac{1-\beta}{\overline{d}}\frac{1}{\delta}\mathbf{p}_u^{K_n}(u) \approx \frac{1-\beta}{\overline{d}}\frac{1}{\delta}\alpha. \tag{3.54}$$

Combining both bounds, we obtain

$$\gamma\delta \geq \frac{\alpha(1-\beta)}{\overline{d}(1-\alpha)}n. \tag{3.55}$$

We proceed similarly for a node $w \notin N_H(u) \cup \{u\}$. By Lemma 3.2.3

$$\mathbf{p}_u^H(w) \leq \frac{(1-\beta)^2}{n-1-\overline{d}}, \tag{3.56}$$

and by Equation 3.52

$$\mathbf{p}_u^H(w) \geq \frac{1}{\delta}\mathbf{p}_u^{K_n}(v) \approx \frac{1-\alpha}{\delta n}. \tag{3.57}$$

Combining these bounds, we get

$$(1-\beta) = \sqrt{\frac{1-\alpha}{\delta}\frac{n-\overline{d}-1}{n}} \geq \sqrt{\frac{1-\alpha}{2\delta}}, \tag{3.58}$$

where in the last step we use, for simplicity, the generous upper bound of $n/2$ for the average degree in a sparse graph.

Plugging this bound on $1 - \beta$ into the earlier inequality 3.55 yields

$$\gamma\delta \geq \frac{\alpha n}{\overline{d}(1-\alpha)}\sqrt{\frac{1-\alpha}{2\delta}} \tag{3.59}$$

$$\gamma\delta^{3/2} \geq \frac{\alpha n}{\overline{d}(1-\alpha)}\sqrt{\frac{1-\alpha}{2}} \tag{3.60}$$

$$\gamma\delta^{3/2} \geq \frac{\alpha n}{\overline{d}\sqrt{2(1-\alpha)}} \tag{3.61}$$

$$(\gamma\delta)^{3/2} \geq \frac{\alpha n}{\overline{d}\sqrt{2(1-\alpha)}} \tag{3.62}$$

$$\gamma\delta \geq (\frac{\alpha}{\overline{d}\sqrt{2(1-\alpha)}})^{2/3} n^{2/3} \tag{3.63}$$

$$\gamma\delta = \Omega(n^{2/3}), \tag{3.64}$$

which establishes the desired bound for the quality. □

This negative result shows that it is not possible to construct sparse multiplicative sparsifiers without significant loss in quality. However, the result is restricted to sparsifiers that do not modify edge weights. This leaves open the possibility that reweighting edges could still yield a useful approximation. As a result, the current bound does not directly apply to the approximate sparsifiers introduced in Section 3.2.2, and we lack a theoretical analysis for them.

4. Implementation and Experiments

In this section, we present and evaluate exact and approximate implementations for computing personalized PageRank sparsifiers. Before that, we briefly introduce the data structures and concepts essential for efficient implementations.

Our implementations and experiments focus on symmetric input matrices, whereas the algorithm proposed by [VCG11] can handle both symmetric and non-symmetric cases. While there has been some initial theoretical work on directed Laplacians (e.g. see [Kyn17]), practical implementations of the required solvers and approximation schemes are largely limited to the symmetric case[1]. However, once such implementations become available, they can be directly integrated and applied to directed graphs.

On one hand, this limitation is unfortunate because PageRank and random walks are properties naturally defined for directed graphs. On the other hand, many real-world graphs are undirected. Additionally, for certain data analysis domains such as graph neural networks, "converting input graphs to undirected ones has become a standard part of the dataset preprocessing pipeline" [RCG+24]. This means that the approach remains applicable to many datasets, and with preprocessing, it can also have applications for directed inputs.

4.1. Data Structures for Vertex Elimination

When computing PPR sparsifiers using our new approach, the most computationally intensive task is computing the Schur complement. Therefore, any efficient implementation of PPR sparsification must also include an efficient method for computing the Schur complement. One strategy is to compute the Schur complement by iteratively eliminating vertices (applying partial Gaussian elimination), which, in graph terms, corresponds to removing a vertex's star and introducing a weighted clique among its neighbors. An efficient data structure for vertex elimination should support the following operations:

1. **Iterating over adjacent nodes/incident edges:**
   We need to efficiently access both incoming and outgoing neighbors. In a matrix representation, this corresponds to accessing the nonzero elements of a row/column. If the graph is undirected the incoming and outgoing neighbors (and their weights) are identical and thus it is enough to have access to one of these two.

[1] The lack of practical implementations is e.g. evident from the absence of a solver for directed graphs in the Laplacians Julia package (`https://juliapackages.com/p/laplacians`), a package that offers some of the fastest algorithms for Laplacian matrices and is continuously updated.

2. **Inserting new edges:**
   Introducing a clique may create edges that did not previously exist, requiring a dynamic data structure capable of efficiently handling edge insertions. These insertions are known as *fill-in*.

3. **Updating existing edge weights:**
   Instead of inserting new edges, adding a clique may modify the weights of existing edges. Efficient updates require fast access to edge (u, v).

With these operations in mind, we aim to find a data structure that allows each elimination step to be performed in time proportional to the size of the introduced clique. More precisely, let G_{i-1} be the graph obtained after eliminating the first $i-1$ vertices. We seek a structure in which the i-th elimination can be performed in time $O(\deg_{G_{i-1}}(v_i)^2)$, where v_i is the vertex eliminated in iteration i.

As the name suggests, the *NodeRemoval* algorithm by [VCG11] is also based on the elimination/removal of nodes, with some additional work. Therefore, the observations we make here will also partially apply to an implementation of their approach.

Storing a graph in a dense matrix structure (*adjacency matrix*) allows edge insertions and updates in $O(1)$, as each entry and thus each edge can be directly accessed. However, retrieving the list of adjacent nodes requires scanning an entire row of the matrix, even though we are only interested in the nonzero entries. This is particularly inefficient for sparse graphs, where an approach that directly depends on nonzero entries can significantly improve performance. Additionally, input graphs often contain up to 100,000 nodes or more, making memory usage a critical concern. Storing the entire matrix in a dense format would quickly become impractical. Therefore, we must use a sparse representation that stores only the nonzero entries, optimizing both memory and computation. As noted by Tarjan [Tar75] in the context of Gaussian elimination, there are two sparse representations that "suggest themselves".

4.1.1. Representation using Hash Tables

When using a dense representation, we typically assume that the vertices are numbered from 1 to $|V|$, allowing us to use a list of size $|V|$ for each node to store and access its incident edges directly based on the node's tag. However, as noted before, this storage method is highly inefficient for sparse graphs, as it requires maintaining a list of size $|V|$ even when a node has only a few neighbors. *Hash tables* [CLRS22] offer a more space-efficient alternative while preserving similar operating times. Instead of using the node value directly as an index, they apply a hash function h to compute an index $h(v)$. This is particularly useful for storing sparse adjacency lists, as it allows mapping to a much smaller range, significantly reducing memory usage. A drawback of using a smaller indexing range is that the hash function may map multiple elements to the same hash value, causing collisions. To handle this, each entry in the hash table stores a bucket, a list of elements that share the same hash. When searching for a value, we must traverse the entire bucket, leading to a worst-case runtime of $O(n)$ for search and deletion [CLRS22]. However, in practice, hash tables are usually much faster and provide constant-time

operations on average by carefully choosing hash functions and rehashing once too many elements have been inserted. Furthermore, we can iterate over all elements in the hash table in $O(n)$ time, for example by maintaining an appropriate pointer structure.

Thus, a graph can be represented by storing the adjacent nodes of each node in a hash table. This graph/matrix format is sometimes referred to as the *Dictionary of Keys* (DOK) representation[2].

An advantage of this approach is its simplicity. First, many programming languages, including `C++`, provide built-in hash table implementations. For example, the standard library includes the data type `unordered_map`[3] which offers the average-case runtime guarantees mentioned earlier. Second, these runtime guarantees allow us to eliminate a node v in the desired $O(\deg(v)^2)$ time on average, enabling us to implement the elimination process directly without modifications. However, a key drawback is that these guarantees hold only on average. Furthermore, when iterating over a node's neighbors, the adjacency information is stored using pointers rather than in contiguous memory. Since modern computer architectures are optimized for sequential memory access, this can lead to poorer cache efficiency and slower traversal compared to array-based representations. Therefore, it will be interesting to evaluate the performance of this graph structure in practice.

4.1.2. Representation using Adjacency Lists

An *adjacency list* represents a graph by storing, for each node v, a list of its neighbors adj_v. If the graph is weighted, each neighbor u is stored along with the corresponding edge weight $w(u, v)$. Conceptually, each entry in the list is a pair consisting of a node ID and its associated weight. This structure can be implemented using either an array-based list or a linked list. The key advantage of adjacency lists is that they allow efficient iteration over a node's neighbors in linear time. Additionally, when using a linked list or a dynamically resizable array, new neighbors can be appended in constant time. However, modifying an existing edge requires searching for it, which takes time proportional to the number of neighbors [CLRS22]. This makes edge updates too slow to achieve the desired running time for a single round of elimination.

The naive approach is therefore insufficient and we need to handle updates differently. Recall that updates occur when a clique is formed during elimination, which typically requires multiple modifications to the same adjacency list adj_v. This process can be understood more intuitively by considering elimination from a matrix perspective: it corresponds to subtracting a scaled row vector. Viewing it this way allows for an optimization that improves performance. The following algorithm efficiently adds two sparse vectors $\mathbf{a}$ and $\mathbf{b}$, assuming access to an auxiliary dense vector $\mathbf{c}$ of size $|V|$, initially set to zero:

1. Load sparse vector $\mathbf{a}$ into $\mathbf{c}$, such that for each $\mathbf{a}(i)$: $\mathbf{c}(\mathbf{a}(i)[\text{"}id\text{"}]) = \mathbf{a}(i)[\text{"}weight\text{"}]$. The vector $\mathbf{a}$ is now available in dense form, where one can directly access each

[2] e.g., by the popular Python package SciPy: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.dok_matrix.html`

[3] `https://en.cppreference.com/w/cpp/container/unordered_map`

entry.

2. Iterate over entries in packed vector **b** and add to dense entries of **c**.
3. Collect non zero elements of **c** and store back into **a**.
4. Reset non-zero elements of vector **c**.

If done correctly, this method achieves updates in time $O(\text{nnz}(\mathbf{a}) + \text{nnz}(\mathbf{b}))$, where $\text{nnz}(\mathbf{x})$ denotes the number of nonzero entries in $\mathbf{x}$. The implementation is most straightforward when **a** contains the same non-zero entries as **b** (requiring only updating the weights), otherwise additional handling for insertions is needed when storing then dense vector back into **a** [DER17]. This approach improves efficiency compared to the naive version but still depends on the degrees of both the eliminated node and its adjacent nodes. Ideally, we want a running time that depends only on the number of neighbors of the eliminated node.

Currently, in the ith-iteration, we eliminate node v_i and update the adjacency lists of all its neighbors that have not yet been eliminated (in the context of Gaussian elimination/matrix factorization this loop ordering is referred to as *right-looking*). However, we can reorder the process: instead of modifying the adjacency lists of v_i's remaining neighbors, we update v_i's adjacency list based on all previously eliminated nodes that were adjacent to it (also known as *up-looking*), computing one row at a time. Since adjacency relationships change during elimination, we must ensure that all nodes processed in previous iterations have been fully updated before proceeding. This reordering ensures that each node's adjacency list is loaded into **c** only once, preventing unnecessary values from being written to the dense vector. As a result, the dominant cost arises from applying the processed rows of previously eliminated nodes. Each such row contains $\deg_{G_{i-1}}(v_i)$ nonzeros and is used to update the rows of all $\deg_{G_{i-1}}(v_i)$ adjacent nodes below it. This yields the desired per-node complexity of $O(\deg_{G_{i-1}}(v_i)^2)$. The trade-off is that, unlike in our standard elimination order, where a processed row is never needed again, we must retain access to all rows throughout the entire elimination process.

As mentioned earlier, the above method for adding two sparse vectors **a** and **b** is simplest when **a** contains at least the same nonzero entries as **b** (or potentially even more). To handle this efficiently, we adopt a strategy commonly used in matrix factorization, splitting the elimination process into two phases:

1. **Symbolic Elimination:** Given elimination order $\pi : V \to \{1, 2, \ldots, |V|\}$, determine the fill-in pattern, meaning process adj_v such that no insertion but only updates will be necessary during elimination by adding entries with weight 0. We achieve this by simulating the elimination process without computing numerical values but only focusing on the adjacency. The symbolic elimination is performed row-wise, but we accelerate it by leveraging the fact that the sparsity pattern of row k is influenced by all previously eliminated rows and only those whose smallest non-zero column index equals k. This approach is based on a graph-theoretic interpretation known as the elimination graph [Liu90].

2. **Numerical Elimination:** In numerical elimination, we perform the actual elimination while leveraging information from the *symbolic elimination* phase. Since the fill-in pattern is determined in advance, dynamic handling of insertions is not required, and only updates need to be applied. Additionally, this enables us to use the *Compressed Sparse Row* (CSR) matrix storage scheme, an alternative sparse matrix data structure that is highly efficient for matrix operations, which was not applicable previously due to its slow insertion times.

While more complicated and complex, this approach allows the most expensive part (numerical elimination) to be performed on continuous memory and might thus be very fast in practice.

4.2. Block Elimination

Earlier we noticed that Schur complement also arises when eliminating an entire block of vertices,

$$\begin{bmatrix} I & 0 \\ -M_{B,S}M_{S,S}^{-1} & I \end{bmatrix} \begin{bmatrix} M_{S,S} & M_{S,B} \\ M_{B,S} & M_{B,B} \end{bmatrix} = \begin{bmatrix} M_{S,S} & M_{S,B} \\ 0 & M_{B,B} - M_{B,S}M_{S,S}^{-1}M_{S,B} \end{bmatrix}. \tag{4.1}$$

Thus, we can also consider an implementation where we solve $M_{B,B} - M_{B,S}M_{S,S}^{-1}M_{S,B}$ directly. This expression involves the inverse matrix $M_{S,S}^{-1}$. However, computing this inverse explicitly is not necessary; instead, we can solve the following linear systems:

$$M_{S,S}^{-1}M_{S,B} = X \tag{4.2}$$

$$M_{S,S}X = M_{S,B} \tag{4.3}$$

Thus, this approach only requires solving $|B|$ linear systems of size $|S| \times |S|$. Furthermore, we can take advantage of the structure of $M_{S,S}$:

- **Sparsity** - In many cases, our input graph and so $M_{S,S}$ will be sparse, making iterative solvers particularly interesting.
- **Special structure** – If the input graph is undirected, then $M_{S,S}$ is a Symmetric Diagonally Dominant (SDD) Z-matrix, for which specialized, efficient solvers exist [GKS23].

4.3. Minimum Degree Ordering

During elimination, a star structure is replaced by a clique. If certain edges did not exist previously, new edges are inserted, a phenomenon known as *fill-in*. Fill-in is undesirable because it increases the elimination graph's density, making subsequent operations slower, as edge updates/insertions happen between pairs of adjacent nodes. Although the final result of the elimination process is independent of the elimination order, the amount

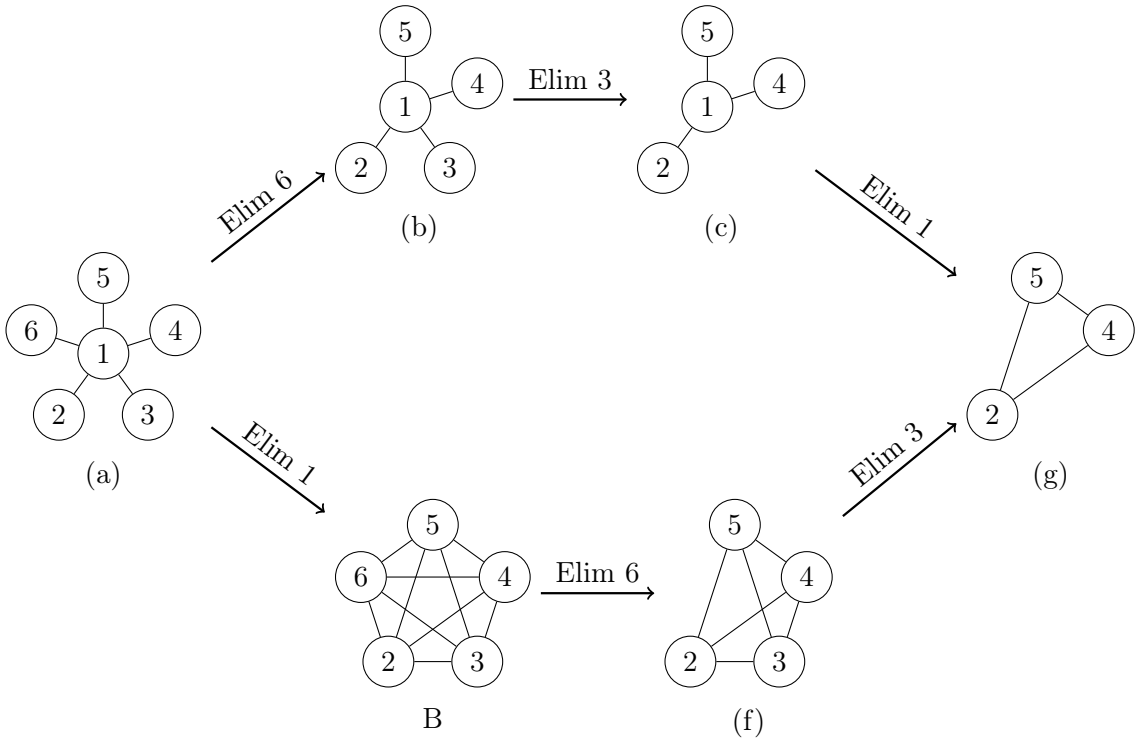

Figure 4.1.: Elimination process using two different elimination orders: (a) Input graph; (b) and (c) show the result after eliminating node 6 or node 1 from (a), respectively; (d) and (e) show the result after eliminating node 3 or node 6 from (b) and (c), respectively; (f) shows the final graph after eliminating node 1 or node 3 from (d) or (e), respectively.

of fill-in that occurs does depend on it. This can be observed in Figure 4.1, where the upper version eliminates nodes in the order 6, 3, and then 1, while the bottom version eliminates nodes 1, 6, and then 3. In the first version, fill-in occurs only after the last node is removed, meaning no additional computational effort is required during earlier steps. In contrast, in the second version, fill-in is introduced already after the first elimination, leading to ten edge insertions, followed by six and three (additional) edge updates in the next two eliminations (already assuming one only operates on a triangular part of the matrix).

Therefore, an elimination-based Schur complement implementation requires an elimination order that minimizes fill-in. However, computing the optimal elimination order (the one with minimum fill-in) is a NP-hard problem [Yan81]. To address this challenge, various heuristics have been developed. In the following, we briefly introduce two common approaches applicable to symmetric structured matrices[4], as discussed in [TW67]. These heuristics are based on the observation that nodes with fewer neighbors create smaller

[4]Symmetric structured matrices have a symmetric nonzero pattern: $M(i,j) \neq 0 \iff M(j,i) \neq 0$

cliques, which in turn result in fewer edges that can be fill-in.

Static Minimum Degree (Tinney 1). This approach computes the unweighted degrees of the input graph and eliminates nodes in ascending order of degree. This method is simple and fast, and the elimination order can be determined solely from the input graph.

Adaptive Minimum Degree (Tinney 2). This approach also computes the degrees of the input graph and eliminates nodes in ascending order of their degree. However, it updates the degrees during elimination to account for fill-in and eliminated edges. Although more complex due to its dynamic nature, where the next elimination node is selected only after completing all prior eliminations, this method typically achieves better fill-in minimization.

In this work, we will evaluate whether the more complex Tinney 2 approach is justified and consider the following two methods for determining the adaptive minimum degree ordering:

4.3.1. Dynamic Minimum Degree Vertex Selection

One option is to determine the next node with minimum degree during the elimination process. For this a bucket-based data structure is appropriate, because a node's degree can only range between 0 and $n-1$. We maintain a list of n buckets, where bucket i points to the head of a doubly linked list storing all nodes with degree i. If a node's degree changes during elimination, it must be removed from its current bucket and inserted into the appropriate new one. Doubly linked lists allow for constant-time insertions and deletions. To quickly extract the node with minimum degree, we maintain a pointer to the bucket containing the smallest degree nodes. When removing a node from this bucket, if the bucket becomes empty, we search for the next non-empty bucket with larger degree. Each elimination decreases the degree of the adjacent nodes by exactly one or increases it. As a result, the minimum degree can also decrease by at most one per eliminated node. This ensures that the total cost of searching for the next non-empty bucket is bounded by $O(n)$ over all eliminations, since we have at most n buckets and can backtrack at most n steps. Given that the total number of eliminations is $O(n)$, the amortized cost of maintaining the minimum degree is constant.

Additionally, a heuristic can be applied to further improve efficiency. Initially, we expect the minimum degree to be low and to gradually increase due to fill-in. However, after k eliminations, the maximum possible degree is bounded by $n-1-k$. This allows for an optimization where, instead of maintaining exactly n buckets, we can use a reduced number and store all nodes with degrees above a certain threshold in the last bucket. This approach minimizes unnecessary movement of high-degree nodes between buckets.

While this option can be easily incorporated into the elimination process, it has two drawbacks. As mentioned earlier, in an undirected graph, it is sufficient to access only the outgoing neighbors, as they are the same as the incoming neighbors. Another property during elimination is that we do not need access to nodes that were eliminated in previous

iterations. Therefore, if the elimination order is known in advance, it suffices to store and operate only on the outgoing edges of nodes that have not yet been eliminated. In matrix notation, this corresponds to only storing the upper triangular part of a symmetric matrix. However, if the order is unknown, we do not know in advance which nodes are required for a given node. In such cases, it is typically best to store and operate on the entire graph, roughly doubling the computational work. Additionally, this requires proceeding with the default (right-looking) elimination loop and does not support the reordering necessary when representing the graph as an adjacency list. For this reason, we also consider the next approach.

4.3.2. Precomputing the Minimum Degree Ordering

An alternative approach is to precompute the elimination order before performing the elimination itself. Fortunately, this is a much simpler problem than executing the elimination and can be significantly accelerated using various optimizations. We still eliminate vertices iteratively and track the minimum-degree vertex using the previously discussed bucket structure. However, we can greatly simplify how we store and manage adjacency information. Below, we provide a rough overview of the most important techniques.

During numerical elimination, processing a clique of size n requires $O(n^2)$ work, as we must store a value for each pair of nodes. However, to determine the minimum degree ordering, we do not need to store numerical values, the clique structure itself is sufficient. Therefore, we can represent a clique using a simple list of its nodes, reducing the storage requirement to $O(n)$ (this storage type is commonly referred to as the *quotient graph*). Initially, we store the adjacency list for each node. When eliminating a node v and creating a clique c, we simply add pointers from v's neighbors to c. When a node is eliminated, we merge its adjacency list with its cliques into a single list and update each neighbor to point to this new clique. To compute the degree of a node, we must account for the unique vertices among its adjacent nodes and the nodes in its cliques. This is the most computationally expensive step, so we aim to avoid computing the degree whenever possible.

Another key optimization leverages *indistinguishable* nodes, which are nodes that share the same adjacency (nodes with identical closed neighborhoods). It can be shown that such nodes remain identical throughout the elimination process. Therefore, we can merge them into a single super node and eliminate them simultaneously, a technique known as *mass elimination*.

Additional optimizations include *incomplete degree updates*, where we exploit the fact that if node u must be eliminated after node v, we do not need to compute or update u's degree until v has been eliminated. Since computing and updating a node's degree is one of the most expensive operations, this reduces computational overhead.

These techniques help accelerate the computation of the minimum-degree ordering. For further details, see [GL89]. Today, additional optimizations are used, such as *multiple elimination* and *approximate minimum degree*. While these methods do not guarantee a true minimum-degree ordering, they are generally faster and usually produce better

orderings, rarely performing worse.

4.4. Experiments Exact Sparsifier

Having introduced the necessary preliminary concepts, we now evaluate the runtime performance of five different implementations for computing exact PPR sparsifiers. Our primary goal is to demonstrate that our algorithms outperform the two methods proposed by Vattani et al. [VCG11] across a range of parameter settings and on both synthetic and real-world datasets of varying sizes.

We begin with an overview of the evaluated implementations. The first two are adapted from [VCG11]:

- ***NodeRemoval***: Since no implementation was provided by the authors, we developed our own based on the published pseudocode. This algorithm operates on a normalized graph and thus is directed even if the input is undirected (unless the graph is regular). However, it is structurally symmetric, allowing us to apply minimum-degree heuristics. The loop structure follows an elimination approach where a star is replaced by a clique. It is unclear whether reordering the elimination loop is possible, as each elimination step additionally introduces edges to the sink node, which are required in the next iteration. This limitation makes adjacency-list-based approaches difficult, as they rely on reordering the elimination loop. Therefore, we use a hash table as the main graph data structure. We aimed for a fair and optimized implementation, avoiding unnecessary operations such as handling two separate graphs, H and G, as seen in the pseudocode. Additionally, the pseudocode includes an infinite sum, which we replace with a closed-form expression:

$$\sum_{t=0}^{\infty}((1-\alpha)w_G(v,v))^t = \frac{1}{\alpha w_G(v,v) - w_G(v,v) + 1}$$

  since the graph is normalized and $|(1-\alpha)w_G(v,v)| < 1$ holds.

- Alternative algorithm ***PPR-MatrixInv***:

  This approach is based on the following equation:

$$W_H^t = \frac{1}{1-\alpha}(I - \alpha P_H^{-1})$$

  where W_H is the walking matrix of sparsifier $H = (V, E, w)$ and an entry $P(i,j)$ corresponds to the personalized PageRank value $\mathbf{p}_i^G(j)$ for $i, j \in V_H$. This version requires the computation of personalized PageRank values on the input graph. According to a recent study, "there is a notable absence of a standardized benchmark or an extensive experimental survey for evaluating the practical performance of existing algorithms" [YWW+24]. For this reason, we chose to implement the standard power iteration algorithm, which has a complexity of $O(km\log(1/\epsilon))$,

where m is the number of edges, k the number of terminal nodes, and ϵ the error tolerance. Nonetheless, alternative algorithms may also be of interest in this context. For inverting an asymmetric dense matrix, we use *Armadillo* [SC25], a linear algebra C++ library.

The remaining three implementations build on the algorithm presented in Section 3.1 but differ in how they compute the Schur complement:

- ***SC-HashTable***: Computes the Schur complement using vertex elimination. The elimination graph is represented as a list of hash tables (see Section 4.1.1). While this method shares similarities with the node removal algorithm, it differs in that our algorithm operates on an undirected graph when the input is undirected. The symmetry of undirected graphs allows us to reduce the computational effort by approximately half in most cases.
- ***SC-AdjList***: Like the previous one, is based on vertex elimination but utilizes an adjacency-based data structure described in Section 4.1.2. As outlined there, the elimination process is divided into two steps. Symbolic factorization follows the method from [GLN94], adapted to compute fill-in only for non-terminal nodes. Numerical factorization is performed in a row-wise order. Since the terminal set is assumed to be small and the resulting sparsifier is typically dense, a dense data structure is used to apply updates from earlier eliminations to the terminal nodes.
- ***SC-BlockElimination***: Computes the Schur complement based on block elimination as described in Section 4.2. It employs a Laplacian iterative solver introduced in [GKS23]. While implementations are available in both `MATLAB` and `Julia`, `MATLAB`'s proprietary nature led us to embed the `Julia` version into our `C++` implementation. This integration may incur slight overhead compared to a fully native solution.

All experiments were conducted on an Ubuntu 24.04.2 LTS system with an Intel(R) Xeon(R) Gold 6130 CPU (32 cores, 64 threads @ 2.10GHz) and 256 GB of memory. We use a restart probability of $\alpha = 0.15$ for all experiments and uniformly sample the terminal nodes from the input graph. The code for all implementations used in our experiments is publicly available in our repository[5].

Since elimination-based algorithms are sensitive to the order in which nodes are eliminated, we begin in Section 4.4.1 by evaluating the impact of different ordering strategies. In Section 4.4.2, we study how the size of the terminal set influences runtime. Finally, in Section 4.4.3, we evaluate the scalability of our methods across a diverse set of graphs.

4.4.1. Impact of Elimination Order

As discussed in Section 4.3, the performance of vertex elimination-based approaches is influenced by the elimination order, which determines the amount of fill-in and thus the overall runtime. We evaluate the following four ordering strategies:

[5] `https://github.com/oleschl/PPR-Sparsifier`

- ***Random***: A random elimination order, used as a baseline for comparison.
- ***Static***: Eliminates nodes in ascending order of their degree in the input graph.
- ***Adaptive1***: Updates node degrees dynamically during elimination, always removing the node with the minimum degree. This is implemented using a bucket-based data structure described in Section 4.3.1.
- ***Adaptive2***: Uses the same ordering as *Adaptive1* but precomputes the elimination order before the elimination process begins (see Section 4.3.2). The implementation is based on the one provided in *Sparspak* (the Waterloo Sparse Matrix Package). To ensure a fair comparison with *Adaptive1*, we do not employ multiple elimination or approximate minimum-degree heuristics. While existing implementations are available, we made slight modifications to ensure that only non-terminal nodes are considered for ordering.

Note that *SC-AdjList* requires the elimination order to be known in advance and therefore does not support the dynamic *Adaptive1* strategy.

We begin with a set of $|K| = 100$ terminal nodes, sampled uniformly at random from the input graph. We expect high running times when using random order; thus, we will begin with a smaller dataset:

- LastFM Asia Social Network [RS20] with 7,624 nodes and 27,806

In addition, we evaluate our method on synthetic data. Specifically, we use *Chimera graphs* [GKS23], which are constructed by transforming and combining various graph primitives such as paths, trees, grids, and random graphs. This construction approach makes them representative of a broad range of graph geometries. We generate 20 Chimera graphs with 5,000 nodes each (using seeds $s \in \{1, 2, ..., 20\}$) and report the worst-case, median, and 75th percentile running times, as we observe high variance across different graph instances. Results are shown in Table 4.1 and 4.2 respectively.

Implementation	**Random**	**Static**	**Adaptive1**	**Adaptive2**
NodeRemoval	2435.72	128.191	13.744	14.352
SC-HashTable	1759.49	53.698	14.684	3.184
SC-AdjList	12.401	0.528	n/a	0.177

Table 4.1.: Comparison of running times (in seconds) on the LastFM network with $|K| = 100$ for different elimination orders and implementations.

Results: We observe in Table 4.1 that *static* ordering performs significantly better than *random*, by roughly a factor of 20 or more, and both adaptive orderings further outperform *static* for all implementations. This confirms that the minimum degree heuristics work well in practice and that the additional overhead of adaptive variants is worthwhile given the resulting performance gains. A similar trend is visible on the synthetic Chimera

Implementation	Metric	Random	Static	Adaptive1	Adaptive2
NodeRemoval	Median	318.49	29.04	2.108	1.876
	75th Percentile	991.05	258.43	27.39	25.73
	Worst	1793.75	998.14	325.17	321.22
SC-HashTable	Median	187.31	8.627	1.211	0.481
	75th Percentile	742.39	136.33	32.70	8.569
	Worst	1690.85	593.11	443.63	203.22
SC-AdjList	Median	0.311	0.0177	n/a	0.0160
	75th Percentile	1.588	0.333	n/a	0.150
	Worst	6.227	2.017	n/a	1.135

Table 4.2.: Comparison of running times (in seconds) on Chimera graphs with $n = 5000$, $|K| = 100$ for different elimination orders and implementations.

graphs (Table 4.2). Although minimum degree based orderings generally perform better, the advantage is less pronounced in worst-case scenarios. This suggests that our heuristics are robust for most graph instances, but there are some for which they bring noticeably smaller improvement over random orderings.

Across all configurations, *SC-AdjList* is consistently the fastest implementation. Although it requires computing the fill-in pattern, and hash table-based approaches have the same theoretical average runtime, array-based lists still prove to be more efficient in practice.

Comparing *NodeRemoval* and *SC-HashTable*, the results for static ordering align with expectations: our approach performs approximately twice as well since it operates on only half of the matrix. For *Adaptive2*, the speedup is slightly larger than expected (except in the worst-case runtime), while for random ordering it is smaller. For *Adaptive1*, where the order is not known in advance, both versions operate on the entire matrix and thus exhibit similar running times.

For *NodeRemoval*, the performance of *Adaptive1* and *Adaptive2* is nearly identical, suggesting that the preprocessing required to compute the ordering has a negligible impact.

Based on these observations, we have decided to proceed with experiments using the ordering computed by the *Adaptive2* strategy for all implementations.

4.4.2. Impact of Terminal Set Size K on Performance

We wanted to investigate how performance is affected by the number of terminal nodes, $|K|$. Using the same datasets as before, we tested with 100, 500, and 1000 terminal nodes, which correspond to 2%, 10%, and 20% of the total nodes in the Chimera graphs (and roughly similar percentages for the real-world LastFM network). These subset sizes are representative of practical use cases for vertex sparsification. The complete results are reported in Tables 4.3 and 4.4, while a representative subset is visualized in Figure 4.2.

Implementation	**|K|=100**	**|K|=500**	**|K|=1000**
NodeRemoval	13.744	31.3565	60.3127
PPR-MatrixInv	1.05149	5.16767	10.204
SC-HashTable	3.184	8.30819	23.2545
SC-AdjList	0.177	0.225004	0.324649
SC-BlockElimination	8.39846	9.53561	11.5739

Table 4.3.: Comparison of running times (in seconds) on the LastFM network for different terminal set sizes $|K| \in \{100, 500, 1000\}$.

Implementation	**Metric**	**|K|=100**	**|K|=500**	**|K|=1000**
NodeRemoval	Median	1.876	4.827	9.849
	75th Percentile	25.73	45.29	69.97
	Worst	321.22	362.53	400.47
PPR-MatrixInv	Median	0.590	2.905	5.912
	75th Percentile	0.660	3.276	6.600
	Worst	1.196	5.834	11.81
SC-HashTable	Median	0.481	1.450	3.848
	75th Percentile	8.569	18.21	38.73
	Worst	203.22	220.56	270.88
SC-AdjList	Median	0.016	0.045	0.080
	75th Percentile	0.150	0.280	0.368
	Worst	1.135	1.447	1.721
SC-BlockElimination	Median	0.218	0.904	1.364
	75th Percentile	0.286	1.072	1.526
	Worst	5.116	5.803	6.584

Table 4.4.: Comparison of running times (in seconds) on Chimera Graph with $|V| = 5000$ for different terminal set sizes $|K| \in \{100, 500, 1000\}$.

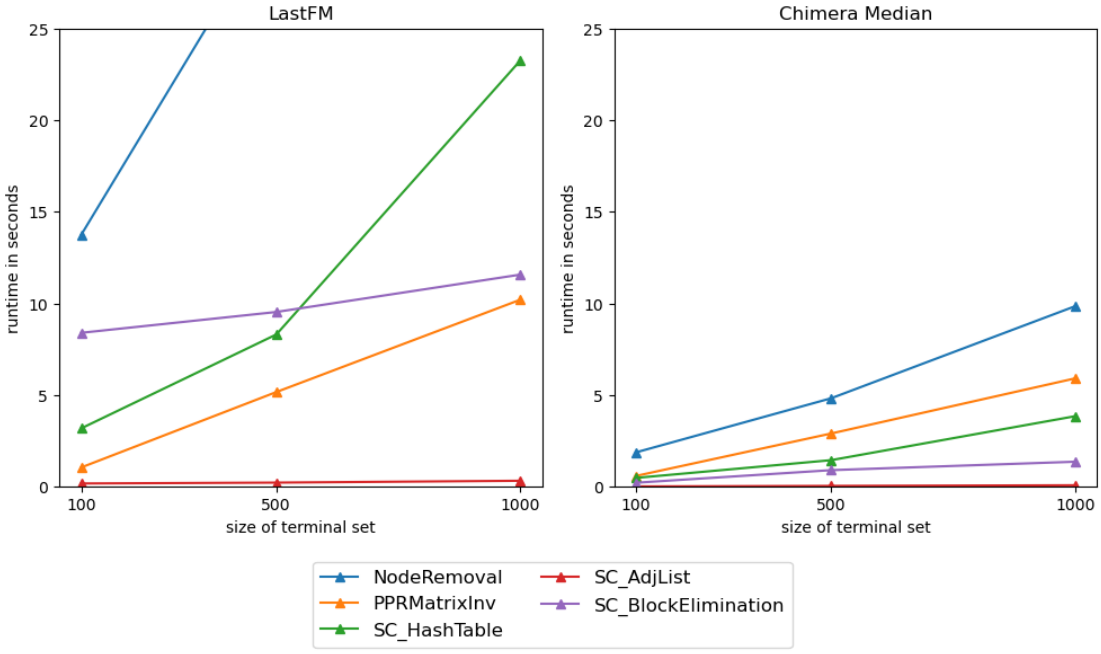

Figure 4.2.: Comparison of runtime across different implementations and sizes of the terminal set K. Right: results on the LastFM graph; left: median values on Chimera graphs (percentiles and worst-case values are omitted for clarity).

Results: It is somewhat surprising that even the elimination-based approaches show a decrease in runtime as the terminal set size decreases, despite the need for more eliminations. This behavior stems from the combination of uniform node sampling and the use of the minimum-degree heuristic. When many nodes are available for elimination, the heuristic can more effectively postpone the removal of high-degree nodes until the graph has already shrunk in size (in terms of number of nodes). Eliminating high-degree nodes later, when the graph is smaller, results naturally in fewer operations and reduced fill-in, thereby lowering the overall computational cost. This leads to an interesting trade-off: when only a few nodes are eliminated (i.e., the terminal set is large), the total number of operations is naturally limited. Conversely, when many nodes are eliminated (i.e., the terminal set is small), the increased number of eliminations can still be performed efficiently in sparse graphs, as the heuristic helps avoid costly eliminations.

For the terminal set sizes we tested, *NodeRemoval* and *SC-HashTable* were consistently the slowest implementations and exhibited significant increases in runtime as $|K|$ grew, roughly doubling or tripling with each step. This can be attributed to the observation above and the overhead of handling more updates and insertions in the hash table.

Among all implementations, *SC-AdjList* remains the fastest across all configurations. However, it also shows a similar multiplicative increase in the median runtime on Chimera graphs as $|K|$ increases. This is expected, as the number of operations on the underlying data structures grows similarly to that of the other elimination-based approaches.

SC-BlockElimination demonstrates a modest growth in runtime with increasing $|K|$, particularly for $|K| = 500$ and $|K| = 1000$ on Chimera graphs, and across all subset sizes on the real-world LastFM network. Much of its runtime appears to be dominated by initial setup costs, such as loading *Julia* functions and computing the approximate factorization required for solving systems of equations. This setup cost does not scale significantly with $|K|$, making the method attractive for both small and large terminal sets. This behavior is also consistent with theoretical expectations: for small $|K|$, fewer but larger systems must be solved, whereas for large $|K|$, a greater number of smaller systems are solved.

Finally, *PPR-MatrixInv* exhibits an approximately linear increase in runtime, as expected, since it must compute a personalized PageRank vector for each terminal. As Vattani et al. [VCG11] note, this method is best suited for small terminal sets. For larger K, it may remain practical when approximate PageRank estimates based on local graph structure are used [ACL06]. Nevertheless, even then the method becomes less viable as K increases, since the main bottleneck eventually shifts to inverting the large, dense PPR matrix.

Another interesting experiment could involve exploring alternative sampling strategies for the nodes in K, beyond simply varying the size. For example, one could focus on removing low-degree nodes or sampling a node along with its immediate neighbors.

4.4.3. Performance Comparison on different sized Datasets

So far, we have focused on smaller datasets. We now turn to evaluating the scalability of the implementations. Since the larger datasets contain significantly more nodes, we also increase the size of the terminal set and use $|K| = 1000$ in these experiments. We use the following real world datasets

- Deezer Europe Social Network [RS20] with 28,281 nodes and 92,752 edges
- GitHub Social Network [RAS19] with 37,700 nodes and 289,003 edges
- Twitch Gamers Social Network [RS21] with 168,114 nodes and 6,797,557 edges
- Pennsylvania Road Network [LLDM09] with 1,088,092 nodes and 1,541,898 edges

In addition, we test again on synthetic Chimera graphs with $50,000$, $100,000$, and $500,000$ nodes. We generate 10 random instances for each size. For each implementation, we begin omitting / aborting experiments once the runtime becomes significantly uncompetitive, specifically, when it is at least a factor of 50 slower than other approaches on the same dataset. The exact values are provided in Tables 4.5 and 4.6, and are visualized in Figures 4.3 and 4.4.

Results: We observe that all elimination-based approaches become uncompetitive on larger datasets. While the adjacency list-based implementation can still handle graphs with a few hundred thousand edges, the *NodeRemoval* and *SC-HashTable* implementations

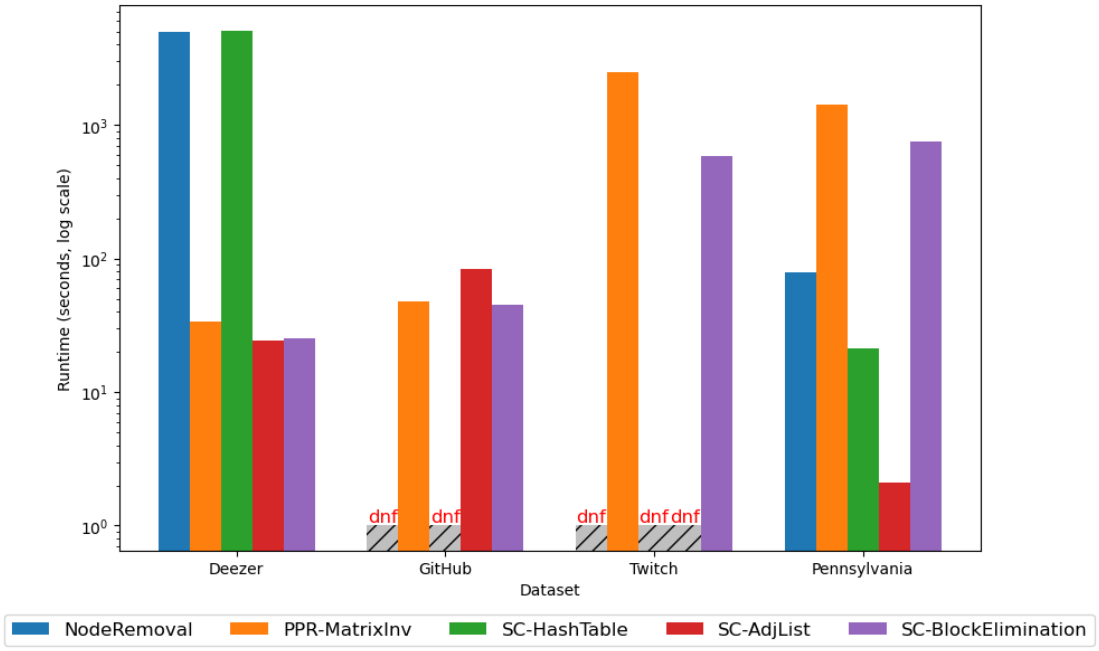

Figure 4.3.: Comparison of runtime across different implementations and real-world graphs of varying sizes.

reach their practical limits at around 100,000 edges. Beyond this point, they are more than 50 times slower and fail to complete within a reasonable time on both Chimera and real-world graphs. For larger datasets, only *PPR-MatrixInv* and *SC-BlockElimination* remain viable, successfully completing within an hour.

This limitation arises from the nature of vertex elimination. Although the minimum-degree heuristic is effective at reducing fill-in and works well up to a certain problem size, beyond that point the elimination graph becomes increasingly dense. In large input graphs, this can occur even when the remaining graph is still of considerable size, leading to substantial slowdowns. In contrast, non-elimination-based approaches avoid this problem by operating directly on the original sparse input graph and producing a dense but small output without creating large, dense intermediate structures.

A notable exception to this trend is the Pennsylvania road network. Road networks are graphs that are nearly planar and exhibit a grid-like structure, with most nodes having a degree of at most four. Eliminating nodes of degree two or less introduces no fill-in, and even eliminations of degree-three or degree-four nodes often result in minimal fill-in due to the geometric locality of their neighbors, who are typically already connected. Consequently, the overall fill-in remains low, and the cost per elimination stays roughly constant for a long portion of the process. This structural advantage makes road networks particularly well-suited for elimination-based methods. In contrast, non-elimination-based methods, although limited to operations on non-zero elements of the input, often incur higher computational overhead (e.g., repeated matrix-vector multiplications), making

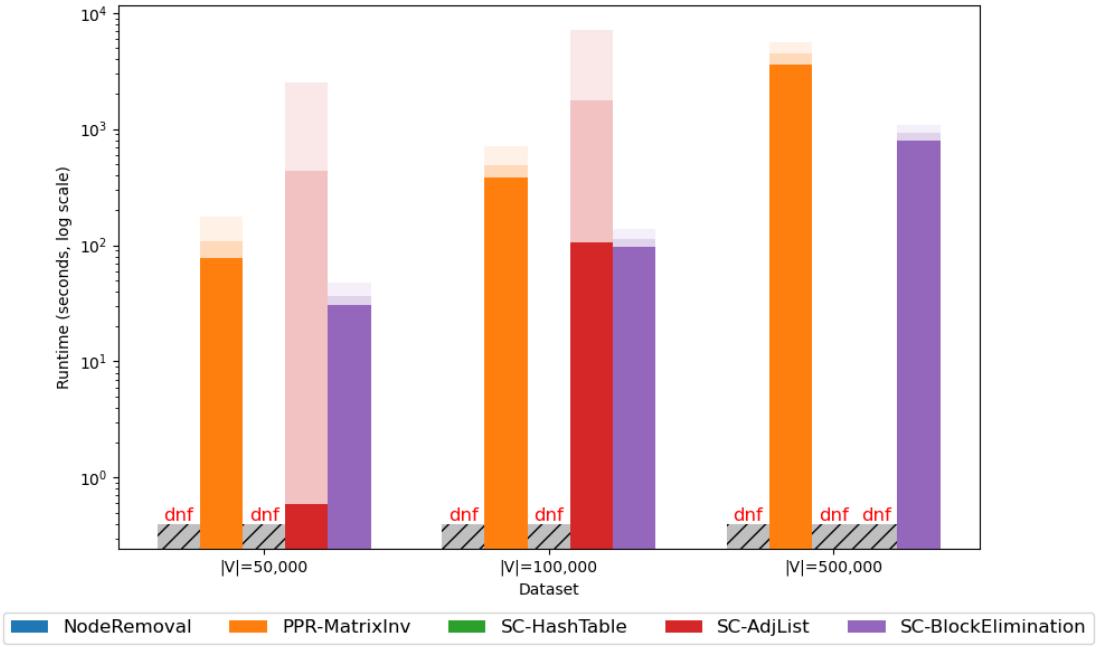

Figure 4.4.: Comparison of runtime across different implementations and Chimera graphs of varying sizes ($|V| \in \{50{,}000,\ 100{,}000,\ 500{,}000\}$). Bars show the median runtime (solid color); the 75th percentile and worst-case runtimes are indicated with the same color at lower opacity.

them slower on such instances.

Overall, our observations align with practical experience in solving linear systems. Elimination-based methods correspond to partial Gaussian elimination. Direct solvers, which perform full Gaussian elimination, are effective up to a certain scale but become prohibitively expensive as problem size increases. Beyond this threshold, iterative methods offer a more scalable solution, particularly for sparse matrices. Our *SC-BlockElimination* method leverages such an iterative solver.

At least when using power iteration, *PPR-MatrixInv* tends to be slower than *SC-BlockElimination* on real-world datasets with a relatively large number of edges, such as the Twitch Gamers Social Network and the Chimera graphs. In contrast, the block elimination approach remains competitive and often the fastest across all tested datasets.

4.4.4. Summary

In summary, our results demonstrate that our algorithms outperform the algorithms proposed in [VCG11]. Hash table-based implementations should generally be avoided: they are slower than the adjacency list variant on small graphs and become infeasible on larger ones. For graphs with up to a few hundred thousand edges, we recommend using *SC-AdjList*. For larger graphs, *SC-BlockElimination* is the preferred choice due to its

Implementation	Deezer	GitHub	Twitch	Pennsylvania
NodeRemoval	5017.53	*dnf*	*dnf*	79.06
PPR-MatrixInv	33.85	47.49	2471.86	1435.08
SC-HashTable	5122.00	*dnf*	*dnf*	21.43
SC-AdjList	24.48	83.50	*dnf*	2.105
SC-BlockElimination	25.41	44.85	590.29	756.57

Table 4.5.: Comparison of running times (in seconds) for different-sized real-world datasets with $|K| = 1000$.

Implementation	Metric	$\|V\| = 50,000$	$\|V\| = 100,000$	$\|V\| = 500,000$
NodeRemoval	Median	*dnf*	*dnf*	*dnf*
	75th Percentile	*dnf*	*dnf*	*dnf*
	Worst	*dnf*	*dnf*	*dnf*
PPR-MatrixInv	Median	78.38	380.12	3599.93
	75th Percentile	109.32	485.60	4520.36
	Worst	174.30	719.10	5617.29
SC-HashTable	Median	*dnf*	*dnf*	*dnf*
	75th Percentile	*dnf*	*dnf*	*dnf*
	Worst	*dnf*	*dnf*	*dnf*
SC-AdjList	Median	0.591	104.70	*dnf*
	75th Percentile	436.79	1779.38	*dnf*
	Worst	2530.92	7100.22	*dnf*
SC-BlockElimination	Median	30.28	96.84	788.25
	75th Percentile	36.80	113.22	923.09
	Worst	47.72	137.70	1088.25

Table 4.6.: Comparison of running times (in seconds) on Chimera graphs of different sizes ($|V| \in \{50{,}000,\ 100{,}000,\ 500{,}000\}$) with $|K| = 1000$.

scalability and efficiency.

While *PPR-MatrixInv* may be a viable option for large graphs (especially when combined with localized personalized PageRank computations) it remains fundamentally limited by the size of the terminal set, as it requires inverting a dense matrix whose size is determined by the terminal set. In most practical settings, *SC-BlockElimination* offers the most robust and scalable performance.

4.5. Experiments Approximate Sparsifiers

In this section, we evaluate two approximate PPR sparsifier implementations, motivated by the findings in Section 3.2.2. These implementations share the same structure as the exact algorithm described in Section 3.1, with the key difference being that they compute an approximate rather than an exact Schur complement. While the computation remains

elimination-based, unlike exact elimination (which introduces a dense clique), these approximate methods sample only a subset of edges to estimate the clique. We consider two approaches: the sampling-based method by Kyng and Sachdeva [KS16], referred to as *RandomClique*, and the elimination-star strategy proposed by Gao et al. [GKS23], referred to as *ElimStar*. We begin with a brief overview of both implementations.

Our `C++` implementation of the ElimStar version is ported from the Julia package `Laplacians.jl`, while the RandomClique variant is implemented directly from pseudocode. Both implementations operate on multigraphs, allowing multiple edges between the same pair of nodes, and they insert new edges during elimination rather than updating the weights of existing ones. A *linked-list*-based data structure is well suited for this setting, as it supports all required operations in constant time and naturally represents multigraphs. For this reason, the `Laplacians.jl` package uses a singly linked list, and we adopt the same structure for our *RandomClique* implementation.

Although the RandomClique method employs a randomized elimination ordering to achieve its theoretical guarantees, we evaluate it in practice using an adaptive minimum-degree heuristic, consistent with the ElimStar implementation.

Both versions use edge splitting, where each edge in the input is replaced with multiple parallel edges. We experiment with different values of the parameter *split* s, which controls the number of such multi-edges. Additionally, the ElimStar approach introduces a *merge* parameter m, which limits the number of parallel edges between two nodes by merging them when the threshold is exceeded. To reduce the parameter space, we follow the recommendation in the source code and set $s = m$ in our experiments.

Otherwise, we use the same experimental setup as in Section 4.4: All experiments were conducted on an Ubuntu 24.04.2 LTS system with an Intel(R) Xeon(R) Gold 6130 CPU (32 cores, 64 threads @ 2.10GHz) and 256 GB of memory. We use a restart probability of $\alpha = 0.15$ for all experiments and uniformly sample 1000 terminal nodes from the input graph. The code for all implementations used in our experiments is publicly available in our repository[6].

We show that both implementations are efficient and scale well to large graphs by demonstrating that their runtimes are competitive with or even outperform those of all exact sparsifier implementations on the same datasets (see Section 4.5.1). To assess the quality of the sparsifiers they produce, we analyze the error norm of the personalized PageRank values, also reported in Section 4.5.1. Finally, to further evaluate their practical utility, we adopt the application setup from [VCG11] and apply both methods to an unsupervised clustering task in Section 4.5.2.

4.5.1. Runtime and Error Norm

We employ the same real-world datasets used in the scalability analysis (see Section 4.4.3) and evaluate the performance of constructing our approximate sparsifiers relative to the fastest runtimes observed therein. To evaluate the quality of the approximations, we compare the input graph G to its approximate sparsifier H by measuring the error

[6] `https://github.com/oleschl/PPR-Sparsifier`

between the resulting personalized PageRank matrices, using the normalized Frobenius norm:

$$\frac{||PPR_G - PPR_H||_F}{||PPR_G||_F} \tag{4.4}$$

The Frobenius norm of a matrix A is defined as:

$$||A||_F = \sqrt{\sum_i \sum_j A(i,j)^2}$$

For nodes u, v in the terminal set K, the matrix entries are given by $PPR_H(u,v) = \mathbf{p}_u^H(v) / \sum_{w \in K} \mathbf{p}_u^H(w)$ and $PPR_G(u,v) = \mathbf{p}_u^G(v) / \sum_{w \in K} \mathbf{p}_u^G(w)$. As a first experiment, we use the error of the induced subgraph as a baseline for assessing quality. Since the induced subgraph serves as a relatively weak reference point, we then partially repeat the experiment using a second baseline, corresponding to the approximate sparsifier presented in [VCG11].

Each sparsifier is created 10 times using different seeds to account for the randomness of our sampling-based algorithms. We report the average running time and error norm along with the standard deviation. For completeness, we also included the results for the Chimera graphs in the appendix, where we observed similar behavior. The results of the first experiment are presented in Figure 4.5, with the exact values provided in the tables in Appendix A.1.1.

Results Experiment 1: We observe that both proposed approaches are significantly faster on all datasets except for the road network, where the exact elimination method using an adjacency list performs exceptionally well (and a linked-list-based structure cannot match the performance). For the Deezer and GitHub datasets, our methods achieve speedups of between 20× and 100× across a broad range of split and merge parameter values. On the Twitch dataset, similar speedups are observed for small parameter values (2 and 3), but the improvement drops to single-digit factors as the parameters increase. We generally observe a near-linear increase in runtime with respect to the split parameter (recall that the initial number of multiedges in the graph is equal to split × the number of input edges), which aligns with the theoretical runtime expectations. This trend is also consistent across datasets, for example, GitHub has roughly three times as many edges as Deezer, and its runtimes are correspondingly about three times higher. When comparing ElimStar and RandomClique, we find that ElimStar is typically faster, but only by a factor of about 2 to 3. This may be due to the limitation on the number of multi-edges imposed by the merge parameter.

In terms of quality, we find that the results are rather strong: the Frobenius norm of the error is small (typically below 0.1 across nearly all settings), indicating that the sparsifier closely approximates the PageRank values on average. We also observe significant improvement over the induced subgraph, even for small parameter values. As the split parameter increases, the error decreases further, improving quality. This improvement is not linear, suggesting that if higher quality is desired, increasing the split parameter continues to help, albeit with diminishing returns.

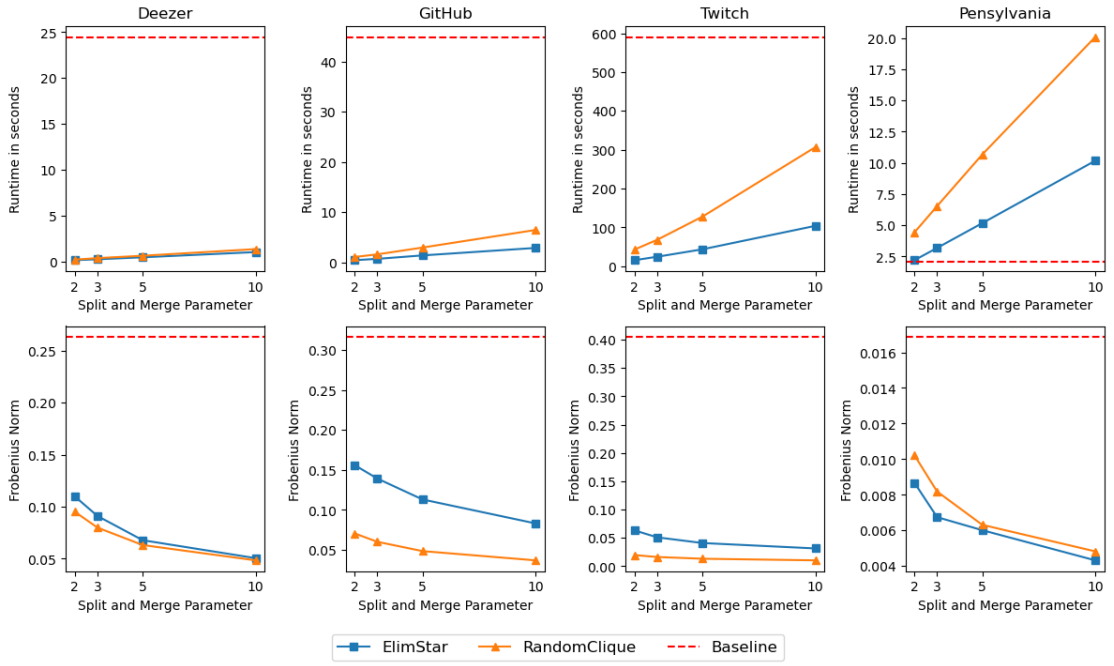

Figure 4.5.: Runtime and quality comparison of our two approximation schemes across different datasets. The runtime baseline corresponds to the fastest observed runtime for computing the exact sparsifier, while the quality baseline reflects the error norm of the induced subgraph.

Interestingly, we observe the lowest error on datasets with a large number of nodes. For example, consider the Pennsylvania dataset, where even the induced subgraph yields a low error. In this case, the graph is reduced from around one million nodes to just $1,000$ in the sparsifier. Due to this drastic reduction and the use of random node sampling, most terminal nodes are likely to be multiple hops apart in the input graph G. This means that a random walker in G, starting at a terminal node, is unlikely to reach another terminal node before either restarting or returning to the original node. As a result, the u-personalized PageRank values in the sparsifier are small for all nodes except u itself. This behavior is similar to that of a graph with no edges at all. This is supported by the observation that, in the exact sparsifier for the Pennsylvania graph, over 97% of the nodes have a combined transition probability from the self-loop and the sink edge exceeding 99% (Recall that transitions to the sink correspond to restarts in the input graph). This indicates that random walks tend to stay at or restart from the personalization node, with the remaining edges having minimal influence. This suggests that it may be beneficial to explore alternative sampling strategies or to increase the number of non-terminal nodes, as the current setting can lead to trivial approximations. It also raises the question of whether additional evaluation metrics beyond the Frobenius norm could better capture meaningful differences in sparsifier quality.

We also observed that the sparsifiers produced by the ElimStar method contain significantly more edges than those from RandomClique, typically between 5 and 10 times as many (see tables in Appendix A.1.1). This difference can arise from several factors, for example, the star sampling process (that always ensures connectivity between neighbors), the effect of the merge parameter, or the fact that RandomClique does not add edges when the same endpoint is sampled multiple times (see Algorithm 3.2.3).

We now briefly describe the second experiment, which was motivated by the observation that the initial quality comparison is somewhat unfair: approximate sparsifiers typically contain significantly more edges than the induced subgraph. To address this, we introduce an additional baseline inspired by the approximation scheme proposed in [VCG11], where an exact sparsifier is computed and then pruned by removing edges with weights below a specified threshold (referred to as the WEIGHTED version in their work). For our experiment, we selected the threshold such that the number of edges in the pruned exact sparsifier matches that of the sparsifier created by RandomClique. We then repeated the previous experiment and compared the Frobenius norm between the two approaches and the input graph to assess their approximation quality. Results are presented in Figure 4.6.

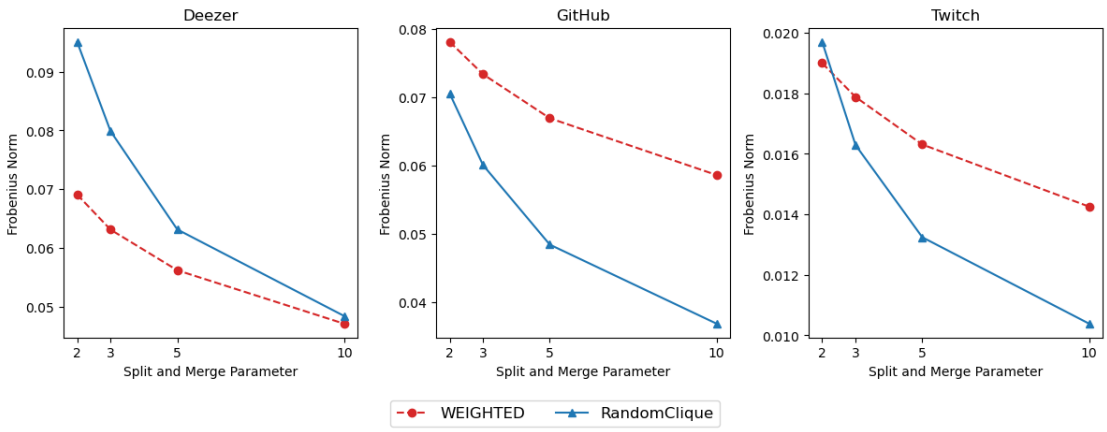

Figure 4.6.: Quality comparison of our RandomClique based scheme across different datasets. The quality baseline corresponds to the error norm between the input graph and the pruned exact sparsifier (WEIGHTED).

Results Experiment 2: We observe that RandomClique performs comparably to, or even slightly better than, the approximation scheme proposed by [VCG11] when the sparsifier is reduced to the same number of edges. This is particularly appealing given that our sparsifiers are significantly faster to compute, as they do not require constructing the exact sparsifier beforehand. However, we note that this comparison is somewhat biased, as it forces the WEIGHTED scheme to match the edge count of our approximate sparsifier. We exclude ElimStar from this evaluation, as the sparsifiers it produces tend to include more edges and provide less pronounced benefits in this setting.

4.5.2. Application: Clustering

While examining the norm provides some indication of quality, it does not directly offer insights into the practical utility of the sparsifiers. To conduct a more meaningful evaluation, we therefore assess their performance in a concrete application. Our goal is to demonstrate that the approximate PPR sparsifiers provide a "good" representation of both the exact sparsifier and the original graph structure for the nodes retained in the sparsifier.

Vattani et al. [VCG11] previously showed that their approximate sparsifiers can serve as effective surrogates for the input graph in unsupervised learning tasks of clustering. Building on this idea, we aim to show that our approximate sparsifiers exhibit similar behavior when used within their clustering framework. Furthermore, we will briefly discuss an alternative clustering approach.

For this evaluation, we restrict our focus to social networks and exclude the Twitch dataset due to its large size.

Approximating the UNWEIGHTED Clustering Subgraph

In [VCG11], the authors describe a clustering approach and show that their constructed subgraph better preserves community structure than the induced subgraph. While they do not use the exact PPR sparsifier directly, they modify it to create what they call the *UNWEIGHTED* version. This is done by selecting a threshold and removing edges with weights below it, so that both the induced subgraph and the pruned sparsifier have the same number of edges. Additionally, they remove the weights to make the graph unweighted. When applying the graph to a clustering algorithm, they further remove edges to the sink node.

To construct the very sparse UNWEIGHTED subgraph, they require the exact sparsifier, which, as we know, is computationally expensive to construct. We aim to show that by using our approximate sparsifier and applying the same edge reduction technique, we obtain a nearly identical graph at a much lower computational cost. To assess graph similarity, we compare the edge sets of both the UNWEIGHTED subgraph and our processed approximate sparsifier. Specifically, we compute the Jaccard index between two edge sets E_1 and E_2, which measures the number of shared edges divided by the total number of edges:

$$J(E_1, E_2) = \frac{|E_1 \cap E_2|}{|E_1 \cap E_2|}$$

To establish a baseline, we also compute the Jaccard score between the edges of the induced subgraph and those of the UNWEIGHTED subgraph. The results are shown in Figure 4.7.

Results: The induced graph shows only around 50% overlap to UNWEIGHTED for all datasets, matching the values reported in [VCG11]. For small values of the split parameter, the results of our sampling based approaches are similar or slightly worse, but with larger split parameters, we observe significant improvements, e.g. nearly a 50%

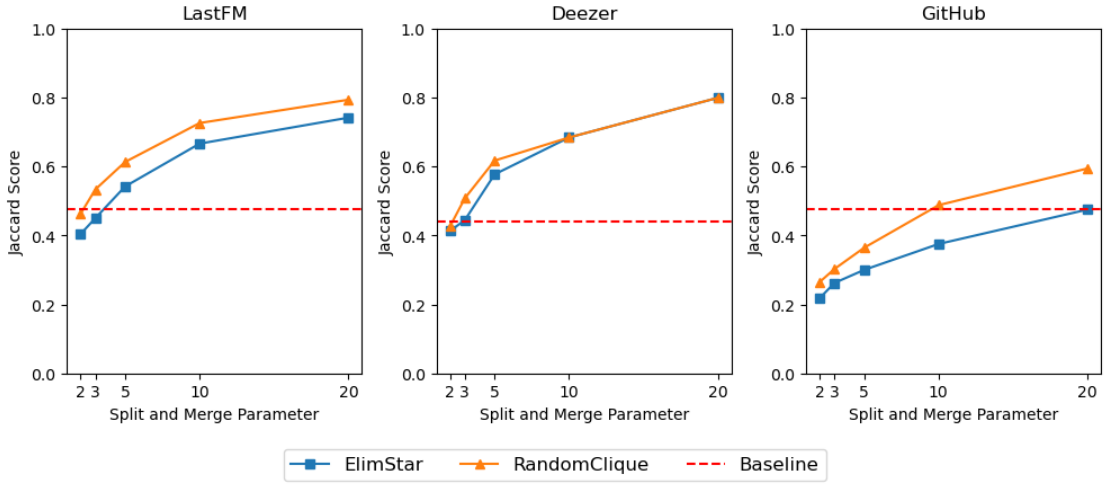

Figure 4.7.: Comparison of graph similarity between UNWEIGHTED and approximate sparsifiers using the Jaccard Index on edge sets for different values of *split* and *merge*.

increase, on the two smaller datasets, LastFM and Deezer. Here they achieve a reasonable overlap of about 80% of the edges. Although the benefit decreases as the split parameter grows further, the results indicate that if higher accuracy is required, it can be achieved by increasing the split parameter, albeit at the cost of increased runtime.

An exception is the GitHub graph, where even with a split parameter of $s = 20$, the overlap for RandomClique remains at only 60%. While this is an improvement over the induced subgraph, the value is still relatively modest. We attribute this to several factors: GitHub is the largest of the three tested graphs, and due to random node sampling, the Induced subgraph contained in our experiment only 173 undirected (or 346 directed) edges, which is quite small. Furthermore, there is an inherent limitation in the sparsification process used for the UNWEIGHTED version. As acknowledged by the authors in [VCG11], their method, due to normalization and threshold rounding, tends to preferentially retain edges adjacent to low-degree nodes. In contrast, for high-degree nodes, where edge weights are more evenly distributed, thresholding can result in nodes losing all outgoing edges. In the exact elimination sparsifier, the graph becomes dense, causing the random walk probabilities to spread across more edges. By contrast, our sampling-based approach introduces fewer edges but reweights them to account for the influence of missing edges. As a result, we expect our approximations to retain edges from high-degree nodes that are likely filtered out in the exact sparsifier when a relatively high threshold is applied to reduce the edge count to just 346. When a higher split value is chosen, the number of edges in the approximate sparsifier increases, potentially distributing the weight more evenly and resulting in performance more similar to that of the UNWEIGHTED subgraph.

Overall, RandomClique tends to perform slightly better than ElimStar, possibly because

it produces sparsifiers with fewer edges, reducing the number of edges that must be removed during the edge reduction step.

In summary, we conclude that for split values around 5–10 or higher, our methods consistently outperform the Induced subgraph and, under certain conditions, provide strong approximations of the UNWEIGHTED graph, achieving up to 80% edge overlap on the tested datasets. We believe that in theses cases, our approximate sparsifier can be effectively integrated into the clustering framework of [VCG11], offering better performance than the induced subgraph while being much more efficient to compute than the UNWEIGHTED version. However, we also note that performance depends on the split parameter and that results for GitHub were less favorable.

We have already noted one limitation of the clustering approach based on the UNWEIGHTED subgraph: it disproportionately penalizes high-degree nodes, potentially reducing them to nodes with no outgoing edges. Additionally, the method employed here explicitly excludes the sink node from the subgraph. As observed, a PPR sparsifier differs from a Schur complement precisely because it retains the sink node and its adjacent edges. Consequently, the subgraph used in this approach is not a true PPR sparsifier but rather an approximate Schur complement. We believe that retaining this additional information is important for accurately preserving the graph's structure. For these reasons, instead of continuing with their clustering approach, we introduce and evaluate an alternative clustering method in the following section.

Alternative Clustering Approach

Ideally, one would expect that personalized PageRank (PPR) sparsifiers provide meaningful information beyond what is captured by the Schur complement alone. One possible reason why [VCG11] chose to exclude the sink node is that their clustering approach is based on graph bisection: once the graph is partitioned, the sink node can belong to only one side of the split, even though it plays an important role in preserving personalized PageRank across both sides of the graph.

Because we wanted to fully leverage the information provided by our approximate PPR sparsifiers but found it challenging to integrate this into a bisection-based algorithm, we decided instead to work with the personalized PageRank vectors, thus indirectly using the entire approximate sparsifier's structure.

Our approach is motivated by the concept of *local graph clustering* [ACL06], which leverages personalized PageRank scores combined with sweep cuts to identify clusters centered around a given source node. This method is particularly appealing for efficiently discovering small, localized clusters without requiring global knowledge of the graph. However, since our sparsifiers are significantly smaller than the original graph, such localized techniques become less necessary.

An alternative approach to using personalized PageRank (PPR) vectors is to compute pairwise distances (e.g., Euclidean distance) between the PPR vectors of different nodes and apply a linkage function to define dissimilarity between node sets. This enables hierarchical clustering and can be used to derive a global clustering of the graph (a similar

setup appears as a pipeline in a framework for evaluating local clustering algorithms[7]). This method aligns well with the intuitive interpretation of personalized PageRank vectors: nodes with similar PPR vectors tend to frequently visit the same regions of the graph and are therefore likely to belong to the same cluster. Since this remains a form of hierarchical clustering, it also allows us to evaluate clustering performance in a way that is comparable to the approach in [VCG11]. Having provided the motivation, we now describe the experimental setup.

We test two approximate sparsifiers: one random clique-based sparsifier with split parameter $s = 10$ (RC_10) and a elimination star-based sparsifier with split and merge parameters $s = m = 10$ (ES_10_10). We compute personalized PageRank vectors for all terminal nodes and use them with cosine similarity and average linkage to perform hierarchical clustering of the graph, as this combination yielded the best results in our preliminary experiments. To assess the similarity between two clusterings, we compare their local structure by computing, for each node, its 10 closest nodes in both clusterings and evaluating the overlap between the two sets (similar to the setup in [VCG11]). Specifically we identify the top 10 most similar nodes based on the *cophenetic* distance, defined as the height in the hierarchy at which two nodes u and v are first merged into the same cluster. We compute these top-10 node sets for both the exact and approximate sparsifiers and compare them using the Jaccard index to quantify their overlap. The mean Jaccard score across all terminal nodes is then reported.

It is worth noting that [VCG11] report using Kendall's τ to assess ranking similarity; however, this measure typically assumes that the rankings are over the same set of elements. In our case, the top 10 nodes from the exact and approximate solutions will likely differ, making Kendall's τ difficult to apply meaningfully. We explored this approach but encountered additional issues (e.g ties) and ultimately believe it is not an appropriate evaluation metric in this context.

To provide a baseline and demonstrate that the task is non trivial, we also perform the same evaluation on the induced subgraph of the input graph. While the induced subgraph serves as a weak baseline, it nonetheless helps contextualize the performance of the sparsifiers. The results are presented in Table 4.7.

Method	**LastFM**	**Deezer**	**GitHub**
Jaccard (Induced subgraph)	0.1468	0.0351	0.0726
Jaccard (ES_10_10)	0.3742	0.2340	0.0889
Jaccard (RC_10)	0.4000	0.2214	0.0874
Relative Improvement (ES_10_10)	154.80%	567.30%	22.30%
Relative Improvement (RC_10)	172.40%	531.40%	20.30%

Table 4.7.: Comparison of clustering similarity between approximate sparsifiers (ElimStar, ES; RandomClique, RC) and the exact sparsifier across different datasets, using the Jaccard Index and relative improvement over induced subgraphs.

[7] `https://github.com/kfoynt/LocalGraphClustering/blob/master/notebooks/find_clusters_social_network.ipynb`

Results: For the LastFM network, we observe a moderate overlap of approximately 40%, indicating a clear correlation with the exact sparsifier. On Deezer, the overlap decreases to 23%, while on the larger GitHub network, the result is notably weak at around 9%.

On the positive side, we observe consistent improvements over the induced subgraph baseline across all datasets, with particularly substantial gains on LastFM and Deezer, where the relative improvement exceeds 100%. However, the absolute performance remains low, and the strong performance observed in norm-based evaluations does not directly translate into similarity in clustering outcomes.

We hypothesize that this is partly due to the combination of random node sampling and substantial node compression. In such settings, random walks in the input graph have a low probability of reaching another terminal node before restarting. As a result, walks in the sparsifier often terminate at the same node they started, and the small personalized PageRank values assigned to other nodes become more difficult to approximate accurately. Further investigation into factors such as the compression ratio, sampling strategies, and the choice of restart probability α would likely yield valuable insights.

Overall, while we observe clear improvements over the induced subgraph baseline and moderate results on certain datasets (such as LastFM), we cannot provide a definitive answer regarding the general effectiveness of the sparsifiers in preserving local graph structure. It remains an open question whether the observed limitations are attributable to the experimental design or the approximate sparsifier themselves.

Finally, we note that this evaluation focuses solely on comparing the clustering similarity between the exact and approximate sparsifiers, without considering the original input graph. We do not elaborate on this here, but comparing to the input graph poses an even greater challenge: whereas the current experiments evaluate whether one vector approximates another of the same dimension, comparisons to the input graph would require identifying a lower-dimensional representation that achieves a comparable approximation, which is closely related to the much more difficult problem of dimensionality reduction.

4.5.3. Summary

We showed that our approximate sparsifier implementations scale nearly linearly with the number of edges in practice, maintaining strong performance even on large-scale graphs. Across the tested datasets, our algorithms consistently ranked among the fastest.

The approximation error of the personalized PageRank values, measured by the Frobenius norm, is low, outperforming the induced subgraph baseline and, for the *RandomClique* variant, closely matching the error of the approximate sparsifier proposed by [VCG11] when matched for the same sparsity level. Notably, the method from [VCG11] requires computing the exact sparsifier as a preprocessing step, whereas our approach does not.

To evaluate how well structural properties are preserved, we applied our approximate sparsifiers to the task of community detection. The results were mixed. While the sparsifiers resemble the UNWEIGHTED graph used in [VCG11] and could potentially be integrated into their clustering framework, offering a trade-off between speed and quality, the observed similarity was inconsistent. Using an alternative clustering method, the

clusterings derived from our approximate sparsifiers outperformed those from the induced subgraph and showed some correlation with the exact sparsifier's output. However, the overlap was only moderate, and similarity was primarily observed between the exact and approximate sparsifiers rather than between the sparsifiers and the original input graph.

Overall, our approximation methods show promise in terms of runtime and error norms, but we cannot yet draw a firm conclusion about their practical utility. Further investigation is needed to identify compelling use cases and determine contexts in which these sparsifiers are most effective.

5. Summary and Future Directions

5.1. Future Directions

The current approach for constructing PPR sparsifiers relies on introducing an additional Steiner node, which may not always be desirable. For instance, when applied to a bisection-based clustering algorithm, this addition introduced complications, and it was unclear how to handle the additional node other than by removing it. A promising direction for future work is to investigate the quality of sparsifiers that can be constructed without Steiner nodes, and to explore methods for building such sparsifiers.

The results of [VCG11] demonstrate that the quality of their sparsifier in applications depends significantly on the sampling scheme, as also briefly discussed in the related work section on graph sampling. In our study, we considered only uniform node sampling and tested a single default value for the restart probability α. It would therefore be valuable to systematically explore how the algorithm's performance varies with different sampling schemes and choices of α.

Another promising direction is to develop implementations of our algorithms for directed graphs and evaluate their performance in this setting. This would primarily require developing efficient methods for computing Schur complements and solving Laplacian systems for asymmetric matrices, which could then be incorporated into our framework.

On the theoretical side, an important open question is to establish approximation guarantees for our approximate variant. More broadly, we demonstrated that without reweighting edges, certain graphs do not admit sparse sparsifiers of good quality. This raises a fundamental question: Is it possible to construct high-quality sparsifiers with provable guarantees when edge reweighting is allowed?

Finally, exploring real-world applications presents another exciting opportunity. While we have investigated graph clustering in this work, the experiments were relatively limited in scope and yielded moderate results. Extending the evaluation to a broader range of settings would help strengthen practical insights. Moreover, applying the approach to compute graph and node embeddings appears particularly promising, as Schur complements have already been successfully used in this context [FGP$^+$20, Kot23]. It would be interesting to explore whether personalized PageRank sparsifiers could be leveraged in such settings to provide useful extensions.

5.2. Conclusion

In this work, we studied graph sparsification, with a focus on vertex sparsifiers that preserve personalized PageRank, as first introduced by [VCG11]. Our main result is a new

algorithm that does not rely on preserving random walks, but instead on preserving the solution to a linear system. This shift in perspective is particularly appealing because it avoids the complexity of random-walk-based arguments and relies only on well-understood linear-algebraic properties of Laplacian matrices and Schur complements.

This insight also enabled several novel implementation strategies. Since computing the Schur complement is the most computationally expensive step in our algorithm, we explored how it can be computed efficiently in practice. We were able to show that our newly developed implementations outperform the original methods in [VCG11]: one, based on vertex elimination using an adjacency list, is particularly efficient for small, sparse graphs (up to a few hundred thousand edges), while another, based on block elimination combined with a Laplacian iterative solver, scales well to larger sparse instances. Both approaches are grounded in the structural insights developed in this work.

We analyzed why computing exact PPR sparsifiers is computationally expensive and explored approximation techniques to accelerate the process. Theoretically, we showed that without reweighting edges, it is generally impossible to obtain high-quality sparse personalized PageRank approximations for dense graphs. Practically, we exploited the structure of the Schur complement to design an approximation algorithm that sparsifies cliques created during vertex elimination by replacing them with sparse substitutes. Using this approximate Schur complement, we constructed approximate PPR sparsifiers. Although this method currently lacks formal theoretical guarantees, our experiments demonstrate its efficiency on large-scale graphs and a low error norm, indicating that personalized PageRank is well approximated on average. Furthermore, we established structural similarity between our approximation and the approximate sparsifiers from [VCG11], while our approach is significantly faster to compute, potentially offering a trade-off between performance and quality. We also demonstrated structural similarity to exact sparsifiers by applying a clustering algorithm based on personalized PageRank vectors, observing moderate correlation in the resulting clusterings and improvements over induced subgraphs. We acknowledge that, while our approximate sparsifier methods are a natural and efficient approach with promising indications of usefulness, a definitive evaluation of their practical impact remains an open question.

In summary, this work establishes efficient methods for both exact and approximate vertex PPR sparsification, with strong empirical results. Our exact sparsifiers provide a scalable and reliable solution, showing clear improvements over previous approaches. The approximate sparsifiers, while lacking formal guarantees at present, demonstrate promising performance and highlight a practical trade-off between computational speed and accuracy. Together, these contributions advance the understanding and practical implementation of vertex sparsification for preserving personalized PageRank.

Bibliography

[ABS+20] Reyan Ahmed, Greg Bodwin, Faryad Darabi Sahneh, Keaton Hamm, Mohammad Javad Latifi Jebelli, Stephen Kobourov, and Richard Spence. Graph spanners: A tutorial review. *Computer Science Review*, 37:100253, 2020.

[ACL06] Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using pagerank vectors. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 475–486, 2006.

[ADD+93] Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete Comput. Geom.*, 9(1):81–100, December 1993.

[AKPW95] Noga Alon, Richard M. Karp, David Peleg, and Douglas West. A graph-theoretic game and its application to the k-server problem. *SIAM Journal on Computing*, 24(1):78–100, 1995.

[Bar96] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 184–193, 1996.

[Bar98] Yair Bartal. On approximating arbitrary metrices by tree metrics. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 161–168, New York, NY, USA, 1998. Association for Computing Machinery.

[BK96] András A. Benczúr and David R. Karger. Approximating s-t minimum cuts in $\tilde{O}$(n2) time. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 47–55, New York, NY, USA, 1996. Association for Computing Machinery.

[BSST13] Joshua Batson, Daniel A. Spielman, Nikhil Srivastava, and Shang-Hua Teng. Spectral sparsification of graphs: theory and algorithms. *Commun. ACM*, 56(8):87–94, August 2013.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[CCPS21] Ruoxu Cen, Yu Cheng, Debmalya Panigrahi, and Kevin Sun. Sparsification of directed graphs via cut balance. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata,*

Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference), volume 198 of *LIPIcs*, pages 45:1–45:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[Chu12] Julia Chuzhoy. On vertex sparsifiers with steiner nodes. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '12, page 673–688, New York, NY, USA, 2012. Association for Computing Machinery.

[CKK+18] Michael B. Cohen, Jonathan Kelner, Rasmus Kyng, John Peebles, Richard Peng, Anup B. Rao, and Aaron Sidford. Solving Directed Laplacian Systems in Nearly-Linear Time through Sparse LU Factorizations . In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 898–909, Los Alamitos, CA, USA, October 2018. IEEE Computer Society.

[CKM+14] Michael B. Cohen, Rasmus Kyng, Gary L. Miller, Jakub W. Pachocki, Richard Peng, Anup B. Rao, and Shen Chen Xu. Solving sdd linear systems in nearly mlog1/2n time. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '14, page 343–352, New York, NY, USA, 2014. Association for Computing Machinery.

[CKP+17] Michael B. Cohen, Jonathan Kelner, John Peebles, Richard Peng, Anup B. Rao, Aaron Sidford, and Adrian Vladu. Almost-linear-time algorithms for markov chains and new spectral primitives for directed graphs. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, page 410–419, New York, NY, USA, 2017. Association for Computing Machinery.

[CLLM10] Moses Charikar, Tom Leighton, Shi Li, and Ankur Moitra. Vertex sparsifiers and abstract rounding algorithms. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 265–274, 2010.

[CLRS22] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, fourth edition*. MIT Press, 2022.

[DER17] I. S. Duff, A. M. Erisman, and J. K. Reid. 18sparse matrices: storage schemes and simple operations. In *Direct Methods for Sparse Matrices*. Oxford University Press, 01 2017.

[DKP+17] David Durfee, Rasmus Kyng, John Peebles, Anup B. Rao, and Sushant Sachdeva. Sampling random spanning trees faster than matrix multiplication. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, page 730–742, New York, NY, USA, 2017. Association for Computing Machinery.

[EEST08] Michael Elkin, Yuval Emek, Daniel A. Spielman, and Shang-Hua Teng. Lower-stretch spanning trees. *SIAM Journal on Computing*, 38(2):608–628, 2008.

[EFL+14] Alessandro Epasto, Jon Feldman, Silvio Lattanzi, Stefano Leonardi, and Vahab Mirrokni. Reduce and aggregate: similarity ranking in multi-categorical bipartite graphs. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, page 349–360, New York, NY, USA, 2014. Association for Computing Machinery.

[EGK+14] Matthias Englert, Anupam Gupta, Robert Krauthgamer, Harald Räcke, Inbal Talgam-Cohen, and Kunal Talwar. Vertex sparsifiers: New results from old techniques. *SIAM Journal on Computing*, 43(4):1239–1262, 2014.

[FGP+20] Matthew Fahrbach, Gramoz Goranci, Richard Peng, Sushant Sachdeva, and Chi Wang. Faster graph embeddings via coarsening. *CoRR*, abs/2007.02817, 2020.

[FHHP19] Wai-Shing Fung, Ramesh Hariharan, Nicholas J. A. Harvey, and Debmalya Panigrahi. A general framework for graph sparsification. *SIAM Journal on Computing*, 48(4):1196–1223, 2019.

[FRT04] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *Journal of Computer and System Sciences*, 69(3):485–497, 2004. Special Issue on STOC 2003.

[GH59] F. R. Gantmacher and K. A. Hirsch. *The theory of matrices*. Chelsea Pub. Co.,, New York :, 1959.

[GHP18] Gramoz Goranci, Monika Henzinger, and Pan Peng. Dynamic effective resistances and approximate schur complement on separable graphs. In Hannah Bast, Grzegorz Herman, and Yossi Azar, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, August 2018.

[GKS23] Yuan Gao, Rasmus Kyng, and Daniel A. Spielman. Robust and practical solution of laplacian equations by approximate elimination, 2023.

[GL89] Alan George and Joseph W.H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 1989.

[GLN94] Alan George, Joseph Liu, and Esmond Ng. Computer solution of sparse linear systems. 1994.

[Gre96] Keith D Gremban. *Combinatorial preconditioners for sparse, symmetric, diagonally dominant linear systems*. PhD thesis, Carnegie Mellon University, 1996.

[HKNR98] Torben Hagerup, Jyrki Katajainen, Naomi Nishimura, and Prabhakar Ragde. Characterizing multiterminal flow networks and computing flows in networks

of small treewidth. *Journal of Computer and System Sciences*, 57(3):366–375, 1998.

[KMP11] Ioannis Koutis, Gary L. Miller, and Richard Peng. A nearly-m log n time solver for sdd linear systems. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, FOCS '11, page 590–598, USA, 2011. IEEE Computer Society.

[Kot23] Vignesh Kothapalli. Randomized schur complement views for graph contrastive learning. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 17580–17614. PMLR, 23–29 Jul 2023.

[KR13] Robert Krauthgamer and Inbal Rika. *Mimicking Networks and Succinct Representations of Terminal Cuts*, pages 1789–1799. 2013.

[KR14] Arindam Khan and Prasad Raghavendra. On mimicking networks representing minimum terminal cuts. *Information Processing Letters*, 114(7):365–371, 2014.

[KS16] Rasmus Kyng and Sushant Sachdeva. Approximate Gaussian Elimination for Laplacians - Fast, Sparse, and Simple . In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 573–582, Los Alamitos, CA, USA, October 2016. IEEE Computer Society.

[Kyn17] Rasmus Kyng. *Approximate Gaussian Elimination*. Phd thesis, ETH Zurich, 2017.

[Law06] G.F. Lawler. *Introduction to Stochastic Processes*. Chapman & Hall/CRC Probability Series. CRC Press, 2nd edition, 2006.

[LF06] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 631–636, New York, NY, USA, 2006. Association for Computing Machinery.

[Liu90] Joseph W.H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1990.

[LLDM09] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6(1):29 – 123, 2009.

[LM04] Amy Langville and Carl Meyer. Deeper inside pagerank. *Internet Mathematics*, 1, 01 2004.

[LM10] F. Thomson Leighton and Ankur Moitra. Extensions and limits to vertex sparsification. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, STOC '10, page 47–56, New York, NY, USA, 2010. Association for Computing Machinery.

[MM10] Konstantin Makarychev and Yury Makarychev. Metric extension operators, vertex sparsifiers and lipschitz extendability. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 255–264, 2010.

[Moi09] Ankur Moitra. Approximation algorithms for multicommodity-type problems with guarantees independent of the graph size. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 3–12, 2009.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[PS89] David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[Räc02] H. Räcke. Minimizing congestion in general networks. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 43–52, 2002.

[Räc08] Harald Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, page 255–264, New York, NY, USA, 2008. Association for Computing Machinery.

[RAS19] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. Multi-scale attributed node embedding, 2019.

[RCG+24] Emanuele Rossi, Bertrand Charpentier, Francesco Di Giovanni, Fabrizio Frasca, Stephan Günnemann, and Michael M. Bronstein. Edge directionality improves learning on heterophilic graphs. In Soledad Villar and Benjamin Chamberlain, editors, *Proceedings of the Second Learning on Graphs Conference*, volume 231 of *Proceedings of Machine Learning Research*, pages 25:1–25:27. PMLR, 27–30 Nov 2024.

[RS20] Benedek Rozemberczki and Rik Sarkar. Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*, page 1325–1334. ACM, 2020.

[RS21] Benedek Rozemberczki and Rik Sarkar. Twitch gamers: a dataset for evaluating proximity preserving and structural role-based node embeddings, 2021.

[RST14] Harald Räcke, Chintan Shah, and Hanjo Täubig. *Computing Cut-Based Hierarchical Decompositions in Almost Linear Time*, pages 227–238. 2014.

[SC25] Conrad Sanderson and Ryan Curtin. Armadillo: An efficient framework for numerical linear algebra. In *Proceedings of the International Conference on Computer and Automation Engineering*, 2025.

[Spi12] Daniel A. Spielman. Spectral graph theory. In Uwe Naumann and Olaf Schenk, editors, *Combinatorial Scientific Computing*, chapter 18, pages 495–524. Chapman and Hall/CRC, 2012.

[Spi19] Daniel A. Spielman. *Spectral and Algebraic Graph Theory Incomplete Draft.* dated December 4, 2019.

[SS11] Daniel A. Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011.

[ST04] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, page 81–90, New York, NY, USA, 2004. Association for Computing Machinery.

[Tar75] Robert E Tarjan. Graph theory and gaussian elimination. Technical report, Stanford, CA, USA, 1975.

[TB97] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997.

[TW67] W.F. Tinney and J.W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967.

[VCG11] Andrea Vattani, Deepayan Chakrabarti, and Maxim Gurevich. Preserving personalized pagerank in subgraphs. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, page 793–800, Madison, WI, USA, 2011. Omnipress.

[Yan81] Mihalis Yannakakis. Computing the minimum fill-in is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, 1981.

[YWW+24] Mingji Yang, Hanzhi Wang, Zhewei Wei, Sibo Wang, and Ji-Rong Wen. Efficient algorithms for personalized pagerank computation: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 36(9):4582–4602, 2024.

A. Appendix

A.1. Data

A.1.1. Approximate Sparsifier Experiments

In this section we report the exact values from the experiments in Section 4.5. Tables A.1, A.2, A.3, and A.4 display the data visualized in Figure 4.5. Tables A.5 and A.6 show the runtimes and error norms for the same experiment, but with synthetic Chimera graphs of different sizes. Table A.7 contains the values corresponding to Figure 4.7.

Implementation	**Metric**	**2/2**	**3/3**	**5/5**	**10/10**
ElimStar	Avg Runtime (s)	0.152	0.251	0.466	1.034
	Runtime Std Dev (s)	0.003	0.005	0.009	0.016
	Avg Norm	0.1098	0.0911	0.0678	0.0505
	Norm Std Dev	0.0029	0.0034	0.0022	0.0010
	Mean Num Edges	101526	142904	202988	295236
	Std Num Edges	776.2	522.0	1124.3	2382.6
RandomClique	Avg Runtime (s)	0.212	0.376	0.633	1.357
	Runtime Std Dev (s)	0.009	0.014	0.018	0.033
	Avg Norm	0.0951	0.0799	0.0632	0.0484
	Norm Std Dev	0.0013	0.0017	0.0016	0.0012
	Mean Num Edges	9200.8	12254.6	18117.0	30848.8
	Std Num Edges	108.8	121.9	163.3	348.3

Table A.1.: **Deezer Network:** Comparison of running times (in seconds) and quality of approximate sparsifiers with $|K| = 1000$.

Implementation	Metric	2/2	3/3	5/5	10/10
ElimStar	Avg Runtime (s)	0.462	0.724	1.408	2.896
	Runtime Std Dev (s)	0.010	0.013	0.052	0.063
	Avg Norm	0.1559	0.1392	0.1130	0.0831
	Norm Std Dev	0.0048	0.0028	0.0035	0.0040
	Mean Num Edges	120551	162869	225805	337476
	Std Num Edges	859.0	1319.4	1006.0	1342.7
RandomClique	Avg Runtime (s)	1.010	1.616	2.973	6.466
	Runtime Std Dev (s)	0.042	0.031	0.060	0.234
	Avg Norm	0.0705	0.0600	0.0484	0.0368
	Norm Std Dev	0.0015	0.0011	0.0008	0.0005
	Mean Num Edges	16681	22558.8	33215.4	56132.4
	Std Num Edges	152.6	182.7	168.7	267.2

Table A.2.: **GitHub Network:** Comparison of running times (in seconds) and quality of approximate sparsifiers with $|K| = 1000$.

Implementation	Metric	2/2	3/3	5/5	10/10
ElimStar	Avg Runtime (s)	15.43	24.28	43.39	103.84
	Runtime Std Dev (s)	0.216	0.401	0.865	6.616
	Avg Norm	0.0632	0.0510	0.0410	0.0314
	Norm Std Dev	0.0042	0.0044	0.0027	0.0037
	Avg Num Edges	443659	512970	599688	722714
	Std Dev Num Edges	1308.4	1716.2	1084.7	1222.8
RandomClique	Avg Runtime (s)	42.88	67.97	127.52	306.55
	Runtime Std Dev (s)	5.792	9.572	15.28	36.19
	Avg Norm	0.0197	0.0163	0.0132	0.0104
	Norm Std Dev	0.0014	0.0009	0.0004	0.0004
	Avg Num Edges	25007.2	34101.6	49925.8	82468.2
	Std Dev Num Edges	135.4	180.8	250.6	362.9

Table A.3.: **Twitch Network:** Comparison of running times (in seconds) and quality of approximate sparsifiers with $|K| = 1000$.

Implementation	Metric	2/2	3/3	5/5	10/10
ElimStar	Avg Runtime (s)	2.203	3.184	5.169	10.17
	Runtime Std Dev (s)	0.000	0.020	0.025	0.083
	Avg Norm	0.0087	0.0067	0.0060	0.0043
	Norm Std Dev	0.0024	0.0014	0.0012	0.0008
	Avg Num Edges	9265.8	12375.2	18019.3	28998.1
	Std Dev Num Edges	139.8	144.1	234.4	306.147
RandomClique	Avg Runtime (s)	4.432	6.534	10.67	20.08
	Runtime Std Dev (s)	0.075	0.047	0.102	0.263
	Avg Norm	0.0102	0.0082	0.0063	0.0048
	Norm Std Dev	0.0015	0.0017	0.0013	0.0012
	Avg Num Edges	2020.2	2027.4	2040	2056.2
	Std Dev Num Edges	5.11642	6.18601	6.39444	7.20802

Table A.4.: **Pennsylvania Network:** Comparison of running times (in seconds) and quality of approximate sparsifiers with $|K| = 1000$.

Implementation	Metric	2/2	3/3	5/5	10/10
RandomClique	Median Runtime (s)	0.336	0.537	0.936	1.934
	75th Percentile Runtime (s)	0.541	0.848	1.478	3.096
	Worst Runtime (s)	1.272	1.980	3.416	7.243
	Mean Norm	0.0513	0.0430	0.0347	0.0269
	Norm Std Dev	0.0107	0.0088	0.0067	0.0047
ElimStar	Median Runtime (s)	0.191	0.318	0.569	1.249
	75th Percentile Runtime (s)	0.451	0.800	1.508	3.426
	Worst Runtime (s)	1.199	2.055	3.846	8.892
	Mean Norm	0.0478	0.0392	0.0314	0.0243
	Norm Std Dev	0.0062	0.0059	0.0044	0.0030

Table A.5.: **Chimera Graph ($|V| = 50{,}000$):** Comparison of running times (in seconds) and quality of approximate sparsifiers with $|K| = 1000$ and different values for *split* and *merge*. For each Chimera instance, the mean runtime was computed over 10 runs, and the median across instances is reported. For the error norm and number of edges, the mean was computed over all runs and all instances.

Implementation	Metric	2/2	3/3	5/5	10/10
RandomClique	Median Runtime (s)	1.144	1.750	3.018	6.283
	75th Percentile Runtime (s)	1.902	2.901	4.924	10.37
	Worst Runtime (s)	2.760	4.187	7.254	15.45
	Mean Norm	0.0320	0.0267	0.0217	0.0170
	Norm Std Dev	0.0092	0.0075	0.0059	0.0044
ElimStar	Median Runtime (s)	1.096	1.914	3.602	7.661
	75th Percentile Runtime (s)	1.649	2.780	5.149	11.50
	Worst Runtime (s)	2.434	4.114	7.687	16.93
	Mean Norm	0.0323	0.0266	0.0210	0.0160
	Norm Std Dev	0.0062	0.0059	0.0044	0.0030

Table A.6.: **Chimera Graph ($|V| = 100{,}000$):** Comparison of running times (in seconds) and quality of approximate sparsifiers with $|K| = 1000$ and different values for *split* and *merge*. For each Chimera instance, the mean runtime was computed over 10 runs, and the median across instances is reported. For the error norm and number of edges, the mean was computed over all runs and all instances.

Configuration	LastFM	Deezer	GitHub
Induced	0.477395	0.440514	0.47548
ElimStar_2_2	0.404224	0.413249	0.21831
ElimStar_3_3	0.450715	0.445161	0.262774
ElimStar_5_5	0.5424	0.577465	0.300752
ElimStar_10_10	0.666379	0.68421	0.375746
ElimStar_20_20	0.741644	0.799197	0.47548
RandomClique_2	0.462822	0.426752	0.265082
RandomClique_3	0.533811	0.508418	0.303202
RandomClique_5	0.613389	0.617329	0.364892
RandomClique_10	0.726052	0.684211	0.488172
RandomClique_20	0.793488	0.799197	0.59447

Table A.7.: Comparison of graph similarity between UNWEIGHTED and approximate sparsifiers using the Jaccard Index on edge sets for different values of *split* and *merge*.