ADAPTIVE ALGEBRAIC MULTIGRID METHODS FOR GRAPHS: ALGORITHMS AND APPLICATIONS

A dissertation

submitted by

Kaiyi Wu

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in

Mathematics

TUFTS UNIVERSITY

August 2023

Adviser: Professor Xiaozhe Hu

Abstract

Graphs are frequently employed to model networks in social science, energy, computer science, and biological applications. In many cases, these applications require the solutions of large-scale linear systems of equations given by the graph Laplacian matrices. In this thesis, we first discuss the fundamentals of graphs and linear systems with graph Laplacians. We present two applications of the graph Laplacian systems: a sophisticated metric on graphs, Diffusion State Distance (DSD), to capture the complicated topological structure of high-dimensional graph data, and a family of semi-supervised graph learning algorithms that propagate labels in such a way as to be consistent with the graph structure. To solve the large-scale graph Laplacian systems arising from these applications, we apply a robust iterative solver, the algebraic multigrid method (AMG) that usually achieves optimal computational complexity in solving such linear systems. We develop efficient, reliable, and computable a posteriori error estimators to aid the AMG solvers. In earlier works, such estimates were computed by solving a perturbed global optimization problem, which could be computationally expensive. We propose a novel strategy to efficiently compute these estimates by constructing a Helmholtz decomposition on the graph based on a spanning tree and the corresponding cycle space. Lastly, we present numerical experiment results for the two applications. We show that DSD distance extracts useful neighborhood information in studying protein-protein interaction networks, as evidenced by the improvements in kNN-based functional classification. For the graph learning problem, we present the analysis of the behavior of Laplace learning, Poisson learning, and p-Laplace learning algorithms in learning cases with a very low label rate.

To my parents, for your constant love and support.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Xiaozhe Hu, for his continuous mentorship and support during my past five years at Tufts University. He is always energetic, approachable, and insightful, and have provided patient guidance and constructive suggestions at every stage of my professional development. I would not have made it this far without him.

I would like to thank the rest of my thesis committee members, Prof. James Adler, Prof. Lenore Cowen, and Prof. Ludmil Zikatanov. I benefited a lot from those collaborations and discussions. A special thank you to Prof. Ludmil Zikatanov for being a tremendous advisor and mentor since my undergraduate study. I am also grateful for the opportunity to collaborate with Prof. James Murphy, Prof. Junyuan Lin, Dr. Yuwen Li, Matthew Werenski, and Kapil Devkota.

I am very lucky to have these amazing friends around me through graduate school. Anca, Chris, Casey, Daniel, Mackenzie, and Marshall, thank you for all the happy memories we have had in the office. I want to acknowledge Shu, Xingchi, and Haohao, for being there with me during the ups and downs since day one.

My family has been an unending source of love and inspiration. My parents, Zhenqiu Wu and Jie Lu, always support me with their unconditional love and understanding. My husband, Yang, has been my backbone and inspired me to be the better version of myself. Last but not least, a big thank you to my 6-day-old daughter Jane who helped me type part of Th_iS maniu.scriphet.

Contents

List of Tables **vii**

List of Figures **ix**

1 Preliminary and Background **2**

1.1 Graphs, Matrix Representations, and Operators 2

1.2 Linear System of Graph Laplacians 5

1.2.1 Diffusion State Distance in Bioinformatics 5

1.2.1.1 Compute DSD 7

1.2.1.2 Approximate DSD with Random Projection 9

1.2.2 Graph Learning Problem 11

1.2.2.1 Laplace Learning and Poisson Learning 13

1.2.2.2 p-Laplace Learning 15

1.3 Solving Graph Laplacian Systems with Algebraic Multigrid 20

1.3.1 Algebraic Multigrid Method 22

1.4 Outline 23

2 Solving Graph Laplacians by Algebraic Multigrid with a Posteriori Error Estimates **24**

2.1 Background on a Posteriori Error Estimates 26

2.1.1 Spanning Tree and Cycle Space on Graphs 26

2.1.2 Previous Results on A Posteriori Error Estimators 27

2.2 Efficient Algorithm for Computing A Posteriori Error Estimator . . . 29

2.2.1 Hypercircle Identity and Error Estimation on Graphs 30

2.2.2 Efficient Evaluation of the Error Estimator 31

2.2.2.1 Computing the Curl-free Component of the Error . . 32
2.2.2.2 Computing the the Div-free Component of the Error 34
2.2.2.3 An Algorithm for A Posteriori Error Estimation with Helmholtz Decomposition 36
2.3 Numerical Results 37
2.3.1 Tests on 2D Uniform Grids 37
2.3.2 Tests on "Real World" Graphs 39
2.3.3 Tests on Building Aggregations for AMG 41
2.4 Alternative Formulation of Error Estimates 43
2.5 Conclusions 45

3 Application: Biological Protein-Protein Interaction Network 47
3.1 Spectral Analysis of Diffusion State Distances 47
3.1.1 Diffusion State Distance and Dimension Reduction 49
3.1.2 Relationship Between Diffusion State Distances and Inverse Laplacians 50
3.1.3 Computational Considerations 51
3.2 Computational and Numerical Experiments on Biological Networks . . 52
3.2.1 Information about the Data Sets 52
3.2.2 Link Prediction 53
3.2.3 Function Prediction 58
3.3 Conclusions and Future Directions 61

4 Application: Graph Learning 62
4.1 Label Classification on Biology Networks 62
4.2 Algorithms for p-Laplace Learning 67
4.2.1 Newton's Method with Homotopy in p 67
4.2.2 Nested Iteration for p-Laplace Learning 68
4.3 Conclusion and Future Work 73

Bibliography 74

List of Tables

2.1 Efficiency of the error estimator on graph Laplacian systems on uniform triangular grids of different sizes. The value of the estimator $\psi(\boldsymbol{\tau})$ is computed by solving (2.13) approximately with 1, 3, and 5 iterations of the overlapping Schwarz method. The CPU time (in seconds) is also shown in the table. 38

2.2 Efficiency of error estimator on graph Laplacian systems arising from "real-world" applications. The value of the estimator $\psi(\boldsymbol{\tau})$ is computed by solving (2.13) approximately with 3 iterations of Schwarz method. The graph types tested are unweighted (u) and weighted (w). 40

2.3 Efficiency of the error estimator on graph Laplacian systems on uniform triangle grids of different sizes. The value of the estimator $\psi(\boldsymbol{v})$ is computed approximately with 1,3, and 5 iterations of the overlapping Schwarz method. The CPU time (in seconds) is also shown in the table. 45

3.1 Number of vertices, number of edges, and average degrees for the largest connected components in the DREAM1 and DREAM2 networks. 53

3.2 Definitions of different heuristic method scores for two nodes $x_i, x_j \in X$ in an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \omega)$ with neighbor sets $\mathcal{N}(x_i)$ and $\mathcal{N}(x_j)$ respectively. 55

3.3 CPU time in seconds used to compute exact DSD and approximate DSD for each DREAM network. The approximate DSD for DREAM1 is computed with the first 6000 eigenvectors and the one for DREAM2 uses the first 1000 eigenvectors. The computation of the approximate DSD is much faster, since only a small number of eigenvectors of $\boldsymbol{P}$ (or equivalently, of $\boldsymbol{N}$) need to be computed. 60

4.1 Function prediction performance on Yeast PPI network with different graph learning algorithms in 5-fold cross-validation. GO labels from the BP hierarchy with a term range of $31-100$ are used. 64

4.2 Function prediction performance on Human PPI network with different graph learning algorithms in 5-fold cross-validation. GO labels from the MF hierarchy with a term range of $31-100$ are used. 65

4.3 Function prediction performance of the graph learning algorithms on both the original Yeast gene co-expression network and the diffused network with 5-fold cross-validation. GO labels from the BP hierarchy with a term range of $11-30$ are used. 66

4.4 Function prediction performance of the graph learning algorithms on both the original Human gene co-expression network and the diffused network with 5-fold cross-validation. GO labels from the MF hierarchy with a term range of $11-30$ are used. 66

4.5 Efficiency of the nested iteration in solving p-Laplace learning on graphs of uniform triangular grids with different sizes. The number of Newton iterations and the converged functional values $J_p^{c_2}(\cdot)$ and $J_p^f(\cdot)$ are reported in the table. 72

List of Figures

1.1 Example of a weighted graph and its graph Laplacian matrix. . . . 3

1.2 Learned functions based on p-Laplacian regularization using different p value with two labeled and 10^5 unlabeled data points on $[0,1]^2$. This example is from [60]. 18

2.1 Fundamental cycle basis 27

2.2 Difference between the true error $\|\boldsymbol{u}-\boldsymbol{v}\|_{\boldsymbol{L}}$ and error estimator $\|\boldsymbol{KGv}-\boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}$ on each edge e. 39

2.3 Representative degree distribution of networks studied in Table 2. *left*: Network ID 33. *middle*: Network ID: 2777 (power law distribution). *right:* Network ID 8 (normal distribution). 40

3.1 *(a), (b), (c):* Comparison of DSD to the different heuristic ranking methods on 100 randomly sampled graphs from DREAM1, each having 400 nodes. On all of the average F1, ROC and precision-recall curves, we see DSD outperforms the heuristic methods. *(d), (e), (f):* Comparison of DSD to the approximate DSD methods on 100 randomly sampled graphs from DREAM1, each having 400 nodes. Limiting the number of eigenvectors used significantly decreased the run time while not substantially degrading the empirical performance. **Note:** The shaded regions demarcate plus and minus one standard deviation of the score. 56

3.2 *(a), (b), (c):* Comparison of DSD to the different heuristic ranking methods on 100 randomly sampled graphs from DREAM2, each having 400 nodes. On all of the average F1, ROC and precision-recall curves, we see DSD is outperformed by the heuristic methods. *(d), (e), (f):* Comparison of DSD to the approximate DSD methods on 100 randomly sampled graphs from DREAM2, each having 400 nodes. Limiting the number of eigenvectors improves results while also lowering computational complexity. **Note:** The shaded regions demarcate plus and minus one standard deviation of the score. . . . 57

3.3 Percentage of the accuracy of function prediction for DREAM1 and DREAM2 networks using the GO Biological Process hierarchy and the Molecular Function hierarchy labels by exact DSD and approximated DSD. The improved function prediction results illustrate the efficiency in denoising the DSD metric by using only the top eigenvectors. We note that both exact and approximate DSD-based function prediction strongly outperform the baseline method. 60

4.1 Number of Newton steps used when we use homotopy in p. 68

Adaptive Algebraic Multigrid Methods for Graphs: Algorithms and Applications

Chapter 1

Preliminary and Background

The scientific study of graphs and networks, such as social networks, computer data networks, and biological networks, is an interdisciplinary field that combines ideas from mathematics, physics, biology, computer science, statistics, social networks, and many other areas [18, 31, 70, 104]. By studying graphs as a data representation of networks, we are able to capture the patterns of interactions within physical or relationship systems, as well as quantify the importance of a particular individual component in the network [104]. We use matrices such as graph Laplacians to represent the graphs, and solving linear systems associated with these matrices helps to answer a variety of research questions of interest.

In this section, We start with the preliminaries of graph Laplacian systems and introduce a family of iterative solvers called algebraic multigrid (AMG). Section 1.1 presents the basics of graphs and related matrices. Section 1.2 includes the discussion of large-scale linear systems of equations given by the graph Laplacian matrices, and two applications that rely on the solutions to such systems. This leads to Section 1.3 to review the background of a robust iterative solver of linear systems called AMG. Lastly, we discuss in Section 1.4 an outline of the rest of the thesis.

1.1 Graphs, Matrix Representations, and Operators

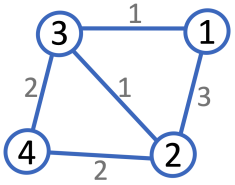
Consider an undirected weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \omega)$, where $\mathcal{V} = \{1, 2, \cdots, n\}$ is the vertex set, $\mathcal{E} = \{e = \{i, j\},\ i, j \in \mathcal{V}, i > j\}$ is the edge set, and $\omega = \{\omega_e\}_{e \in \mathcal{E}}$ is the set of edge weights. Here, the weights are assumed to be positive, i.e., $\omega_e > 0$, and we take all the edge weights to be 1 for unweighted graphs. We only consider undirected graphs here and that is why we have fixed $i > j$ for every $e = \{i, j\} \in \mathcal{E}$. Thus $\{1, 2\}$ cannot be an edge in our graph, while $\{2, 1\}$ could be in $\mathcal{E}$. This notation also assumes that there are no self-loops and multiple edges.

Denote $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$. The degree of a vertex $i \in \mathcal{V}$ is denoted as d_i, which is the number of edges connected to i. The weighted degree is defined as $\delta(i) := \sum_{j \in \mathcal{V}, e=\{i,j\} \in \mathcal{E}} \omega_e$.

Next, we introduce several matrices related to the graphs. The adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ and the weighted degree matrix $\boldsymbol{D} \in \mathbb{R}^{n \times n}$ are defined as follows,

$$A_{ij} = \begin{cases} \omega_e, & \text{if } e = \{i,j\} \in E, \\ 0, & \text{otherwise.} \end{cases} \qquad D_{ij} = \begin{cases} \delta_i, & i = j, \\ 0, & i \neq j. \end{cases} \tag{1.1}$$

Then the weighted graph Laplacian of $\mathcal{G}$ is defined as $\boldsymbol{L} := \boldsymbol{D} - \boldsymbol{A}$. See Figure 1.1 for an example of a weighted graph with 4 nodes and 5 edges, along with its graph Laplacian matrix.



$$\boldsymbol{L} = \begin{pmatrix} 4 & -3 & -1 & 0 \\ -3 & 6 & -1 & -2 \\ -1 & -1 & 4 & -2 \\ 0 & -2 & -2 & 4 \end{pmatrix}$$

Figure 1.1: Example of a weighted graph and its graph Laplacian matrix.

Alternatively, we can define the graph Laplacian as a bilinear form as follows. Let $\mathscr{V} = \mathbb{R}^n$ and $\mathscr{W} = \mathbb{R}^m$ be the vertex space and edge space, respectively. The inner products on vertex space and edge space are defined as:

$$(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{v}^\top \boldsymbol{u}, \quad \forall\ \boldsymbol{u}, \boldsymbol{v} \in \mathscr{V},$$
$$(\boldsymbol{\tau}, \boldsymbol{\phi}) = \boldsymbol{\phi}^\top \boldsymbol{\tau}, \quad \forall\ \boldsymbol{\tau}, \boldsymbol{\phi} \in \mathscr{W}.$$

The weighted graph Laplacian matrix $\boldsymbol{L} \in \mathbb{R}^{n \times n}$ can be defined via the bilinear form:

$$(\boldsymbol{L}\boldsymbol{u}, \boldsymbol{v}) := \boldsymbol{v}^\top \boldsymbol{L}\boldsymbol{u} = \sum_{e=\{i,j\} \in \mathcal{E}} \omega_e (\boldsymbol{u}_i - \boldsymbol{u}_j)(\boldsymbol{v}_i - \boldsymbol{v}_j), \quad \forall\ \boldsymbol{u}, \boldsymbol{v} \in \mathscr{V}.$$

The normalized graph Laplacian $\boldsymbol{N}$ is defined as:

$$\boldsymbol{N} := \boldsymbol{D}^{-\frac{1}{2}} \boldsymbol{L} \boldsymbol{D}^{-\frac{1}{2}}. \tag{1.2}$$

We note that $\boldsymbol{L}$ and $\boldsymbol{N}$ are both symmetric positive semi-definite (SPSD) matrices. For connected graph, $\mathcal{G}$, the graph Laplacian $\boldsymbol{L}$ has an eigenvalue of 0, and the corresponding eigenvector is a constant vector $\mathbf{1} = [1, 1, \cdots, 1]^\top$. The 0 eigenvalue of $\boldsymbol{N}$ has the corresponding eigenvector $\boldsymbol{d}$ defined as,

$$\boldsymbol{d} := \frac{\boldsymbol{D}^{\frac{1}{2}}\mathbf{1}}{\sqrt{\delta_{\text{total}}}}, \tag{1.3}$$

where the total weighted degree is $\delta_{\text{total}} := \sum_{i \in \mathcal{V}} \delta(i)$. The eigenvector $\boldsymbol{d}$ is normalized such that $\|\boldsymbol{d}\|_2 = 1$.

We associate with each graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{\omega})$, the discrete gradient operator (or edge-node incidence matrix), $\boldsymbol{G} \in \mathbb{R}^{m \times n} : \mathscr{V} \to \mathscr{W}$, and the edge weight matrix, $\boldsymbol{K} \in \mathbb{R}^{m \times m} : \mathscr{W} \to \mathscr{W}$. They are defined as the following: for each edge $e = \{i, j\} \in \mathcal{E}$,

$$\begin{aligned} (\boldsymbol{G}\boldsymbol{v})_e &= \boldsymbol{v}_i - \boldsymbol{v}_j, \quad \forall\ \boldsymbol{v} \in \mathscr{V}, \\ (\boldsymbol{K}\boldsymbol{\tau})_e &= w_e \boldsymbol{\tau}_e, \quad \forall\ \boldsymbol{\tau} \in \mathscr{W}. \end{aligned} \tag{1.5}$$

The adjoint of $\boldsymbol{G}$, denoted by $\boldsymbol{G}^\top : \mathscr{W} \to \mathscr{V}$, is the discrete divergence operator (or node-edge incidence matrix) on the graph,

$$(\boldsymbol{G}\boldsymbol{u}, \boldsymbol{\tau}) = (\boldsymbol{u}, \boldsymbol{G}^\top \boldsymbol{\tau}), \quad \forall\ \boldsymbol{u} \in \mathscr{V},\ \forall\ \boldsymbol{\tau} \in \mathscr{W}. \tag{1.6}$$

By direct computation, the following identity holds true,

$$(\boldsymbol{L}\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{K}\boldsymbol{G}\boldsymbol{u}, \boldsymbol{G}\boldsymbol{v}).$$

Thus, we can write,

$$\boldsymbol{L} := \boldsymbol{G}^\top \boldsymbol{K} \boldsymbol{G}. \tag{1.7}$$

Based on this definition of the graph Laplacian $\boldsymbol{L}$, it is straightforward to verify that,

$$\|\boldsymbol{K}\boldsymbol{G}\boldsymbol{u} - \boldsymbol{K}\boldsymbol{G}\boldsymbol{v}\|^2_{\boldsymbol{K}^{-1}} = \|\boldsymbol{u} - \boldsymbol{v}\|^2_{\boldsymbol{L}}, \quad \forall\ \boldsymbol{u}, \boldsymbol{v} \in \mathscr{V},$$

where $\|\boldsymbol{\tau}\|^2_{\boldsymbol{K}^{-1}} = (\boldsymbol{\tau},\boldsymbol{\tau})_{\boldsymbol{K}^{-1}} := (\boldsymbol{K}^{-1}\boldsymbol{\tau},\boldsymbol{\tau})$, $\forall\ \boldsymbol{\tau} \in \mathscr{W}$, and $\|\boldsymbol{v}\|^2_{\boldsymbol{L}} = (\boldsymbol{v},\boldsymbol{v})_{\boldsymbol{L}} := (\boldsymbol{L}\boldsymbol{v},\boldsymbol{v})$, $\forall\ \boldsymbol{v} \in \mathscr{V}$.

1.2 Linear System of Graph Laplacians

In many cases, the applications involving graphs and networks require the solution of large-scale linear systems of equations given by the graph Laplacian matrix [45,90,98, 114,131]. Such linear systems have also been the key to developing link-based ranking algorithms for web searching queries [91,109,130] or recommendation systems [61,94]. Exploiting the properties of graph Laplacians to achieve dimension reduction in high dimensional data representation, researchers have produced fruitful results in image classification [64, 138], representation learning [67], and clustering [10, 106]. In the numerical solutions of partial differential equations (PDEs), the stiffness matrices arising from the finite-element or finite-difference method also take the form of graph Laplacians as discussed in [135]. Therefore, it is important to develop efficient and robust methods for solving graph Laplacian systems.

We give two concrete examples of the applications of graph Laplacian systems in Section 1.2.1 and Section 1.2.2. We first introduce the background and motivation of the applications, and show how solving the linear systems with graph Laplacians is the key to tackling these two problems. The numerical results are demonstrated in Chapter 3 and Chapter 4, respectively.

1.2.1 Diffusion State Distance in Bioinformatics

In this subsection, we introduce the Diffusion State Distance(DSD), a metric on the graph that compares points using a data-driven diffusion process. DSD provides a powerful tool for learning the underlying structure of high-dimensional data. We show that computing this DSD metric is equivalent to solving a linear system $\boldsymbol{N}\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{N}$ is the normalized graph Laplacian.

In humans, as well as several well-studied model organisms, biologists study different types of gene-gene or protein-protein association networks, where the vertices

represent genes or proteins, and two proteins are connected by an edge based on different criteria depending on the network. For example, the classical protein-protein interaction networks connect two proteins if there is experimental evidence that they bind in the cell, with an edge weight that corresponds to either the confidence in the experimental evidence, or the predicted strength of the interaction. Other networks connect two genes if they are typically expressed in the same human tissues, or if their genetic sequence is sufficiently similar. A classical and well-studied problem in Bioinformatics involves leveraging neighborhood information in protein-protein interaction (PPI) networks to predict protein functional labels. Some vertices are partially labeled with one or more functional labels representing what is known about their functional role in the cell. These functional labels are derived from some biological ontology (most commonly GO, the Gene Ontology [46]). Protein-protein association networks can assist in making predictions for functional roles of unknown proteins, since proteins with similar functions should cluster (under the appropriate metric) in the network.

The PPI networks, as well as social networks, are known as "small-world" networks [104]. These networks have small diameters, and most of the nodes in the network are close to each other when measured with the shortest-path distance, which is referred to as the "tie-in-proximity" phenomenon. The shortest-path distance between a pair of nodes is defined as the smallest number of edges connecting two nodes. Besides the "tie-in-proximity" problem, the shortest path distance also failed to capture some other underlying graph features. For example, the fact that two nodes that have the shortest path distance equal to 2 can convey something very different than two other nodes that share the same distance. While all being distance two away from each other, two nodes with many low-degree common neighbors should be thought of as "more similar" than nodes with few low-degree common neighbors. At the same time, such nodes should also be thought of as "more similar" than two nodes with high-degree common neighbors(known as the hub nodes). In biological networks, the hub nodes tend to be involved in the general machinery of the cell, as an example, chaperone proteins (proteins that help other proteins fold

correctly) or proteins involved in translation [38], and are not typically functionally related to their interaction neighbors.

Diffusion state distance (DSD) was first proposed in the context of PPI networks [38], and generalized to graphs with weighted edges in [36]. It is a finer-grained distance metric designed to capture complex structures in a manner robust to high-degree nodes that ruin the discriminative ability of the classical shortest path metrics in small-world networks. [36, 38] show that replacing the ordinary shortest-path distance with DSD lead to dramatic improvements when using network information to predict protein functions. DSD is further developed based on the classical potential theory for applications with general finite irreducible Markov chains [96]. [76, 77] provides theoretical analysis for the convergence of DSD. Furthermore, [90] proposes to accelerate the computation of DSD with unsmoothed aggregation algebraic multigrid, and formulates a faithful approximation of DSD by using random projections based on the Johnson–Lindenstrauss lemma.

Once the DSD is obtained on the biological networks, we can make use of this distance metric for many downstream graph learning tasks like function prediction or link prediction in computational biology, which is elaborated on in Chapter 3.

1.2.1.1 Compute DSD

The graph-diffusion-based DSD is motivated by random walks on the graph. We first define the one-step transition matrix $\boldsymbol{P}$ on the graph $\mathcal{G}$. For simple connected and undirected graph, $\boldsymbol{P}$ is defined as,

$$\boldsymbol{P} \coloneqq \boldsymbol{D}^{-1}\boldsymbol{A} = \begin{cases} \frac{w_e}{\delta(i)}, & \text{if } e = \{i, j\} \in \mathcal{E}, \\ 0, & \text{otherwise.} \end{cases} \tag{1.8}$$

Note that $\boldsymbol{P}$ is row stochastic and irreducible. If $\boldsymbol{P}$ is also aperiodic, then $\boldsymbol{P}$ admits a unique stationary distribution $\boldsymbol{\pi} \in \mathbb{R}^n$, defined as,

$$\boldsymbol{\pi} = \delta_{\text{total}}^{-1}\boldsymbol{D}\mathbf{1}, \tag{1.9}$$

where $\boldsymbol{\pi}$ is the left eigenvector of $\boldsymbol{P}$, i.e. $\boldsymbol{\pi}^\top P = \boldsymbol{\pi}^\top$. In addition, $\boldsymbol{\pi}$ is normalized as $\boldsymbol{\pi}^\top \mathbf{1} = 1$.

Given the transition matrix, $\boldsymbol{P}$, we define DSD as in [38]. We associate each vertex i with a vector $\boldsymbol{h}_i^q \in \mathbb{R}^n$ such that $(\boldsymbol{h}_i^q)_j$ represents the expected number of times for a random walk to start from vertex i and visit vertex j within q steps. It is easy to see that,

$$\boldsymbol{h}_i^q := (\boldsymbol{I} + \boldsymbol{P}^\top + (\boldsymbol{P}^\top)^2 + \cdots + (\boldsymbol{P}^\top)^q)\boldsymbol{\psi}_i,$$

where $\boldsymbol{I}$ is the identity matrix and $\boldsymbol{\psi}_i$ is the i-th column of the identity matrix. Then, the q-th step DSD between the vertex i and the vertex j is defined as

$$\mathrm{DSD}^q(i,j) := \|\boldsymbol{h}_i^q - \boldsymbol{h}_j^q\|_p,$$

with $\|\cdot\|_p$ being the standard ℓ_p-norm. Then, the diffusion state distance is defined by letting $q \to \infty$, if this limit exists,

$$\mathrm{DSD}(i,j) := \lim_{q\to\infty} \mathrm{DSD}^q(i,j). \tag{1.10}$$

The following convergence theorem has been shown in [38, 90].

Theorem 1.2.1 (Convergence for DSD [38, 90]) *For a connected graph $\mathcal{G}$ whose one-step transition matrix $\boldsymbol{P}$ is diagonalizable and ergodic as a Markov Chain,* $\mathrm{DSD}^q(i,j)$ *converges as $q \to \infty$,*

$$\mathrm{DSD}(i,j) := \lim_{q\to\infty} \mathrm{DSD}^q(i,j) = \|(\boldsymbol{I} - \boldsymbol{P}^\top + \boldsymbol{W}^\top)^{-1}(\boldsymbol{\psi}_i - \boldsymbol{\psi}_j)\|_p, \tag{1.11}$$

where,

$$\boldsymbol{W} := \mathbf{1}\boldsymbol{\pi}^\top, \tag{1.12}$$

is the Perron projection of $\boldsymbol{\pi}$ defined in (1.9).

The diffusion state $\boldsymbol{X} \in \mathbb{R}^{n\times n}$ is defined as,

$$\boldsymbol{X} := (\boldsymbol{I} - \boldsymbol{P}^\top + \boldsymbol{W}^\top)^{-1}. \tag{1.13}$$

In practice, to compute DSD between all the pairs of vertices, $\boldsymbol{X}$ is pre-computed and stored. Then DSD between node i and node j is computed as $\mathrm{DSD}(i,j) = \|\boldsymbol{x}_{ij}\|_p$, where $\boldsymbol{x}_{ij} = (\boldsymbol{I}-\boldsymbol{P}^\top+\boldsymbol{W}^\top)^{-1}(\boldsymbol{\psi}_i-\boldsymbol{\psi}_j)$. We note that the matrix $\boldsymbol{I}-\boldsymbol{P}^\top+\boldsymbol{W}^\top$ is dense even if the graph $\mathcal{G}$ itself is sparse, and as a result, in practice, a $\mathcal{O}(n^3)$ computational cost is usually required to invert for $(\boldsymbol{I} - \boldsymbol{P}^\top + \boldsymbol{W}^\top)^{-1}$, which is computationally expensive or even infeasible. Therefore, an alternative formulation of DSD is used in the implementation to improve overall efficiency.

Theorem 1.2.2 ([90]) *For a simply connected undirected graph $\mathcal{G}$ with $\boldsymbol{D}, \boldsymbol{N}, \boldsymbol{P}$, and $\boldsymbol{W}$ defined by* (1.1), (1.2), (1.8), *and* (1.12), *we have,*

$$(\boldsymbol{I} - \boldsymbol{P}^\top + \boldsymbol{W}^\top)^{-1} = \boldsymbol{D}^{\frac{1}{2}}(\boldsymbol{N}^\dagger + \boldsymbol{d}\boldsymbol{d}^\top)\boldsymbol{D}^{-\frac{1}{2}},$$

where $\boldsymbol{N}^\dagger$ is the pseudoinverse of $\boldsymbol{N}$.

With Theorem 1.2.2, we can compute the DSD by solving for the pseudoinverse of $\boldsymbol{N}$ instead of inverting the dense matrix $\boldsymbol{I} - \boldsymbol{P}^\top + \boldsymbol{W}^\top$. Dealing with the normalized graph Laplacian $\boldsymbol{N}$ is computationally easier since $\boldsymbol{N}$ is symmetric and positive semi-definite. Moreover, $\boldsymbol{N}$ is sparse if the graph $\mathcal{G}$ is sparse. Efficient iterative solvers, such as AMG, can be applied to solve for linear systems with $\boldsymbol{N}$ at a modest computational cost. For sparse $\boldsymbol{N}$, comparing to the original formulation to invert $\boldsymbol{I} - \boldsymbol{P}^\top + \boldsymbol{W}^\top$ with $\mathcal{O}(n^3)$ cost, the computational cost with the reformulation by Theorem 1.2.2 is reduced to $\mathcal{O}(n)$ with AMG solvers. We will discuss the family of AMG algorithms in detail in Section 1.3.

1.2.1.2 Approximate DSD with Random Projection

Note that the i-th column of the diffusion state $\boldsymbol{X}$ can be viewed as a new coordinate presentation of the vertex i in $\mathbb{R}^n$. When n is large, the coordinates of

the new representation are still in high dimension. Therefore, we are motivated to use dimension reduction to further reduce the computational cost, besides the reformulation trick mentioned in Theorem 1.2.2. In [90], based on the well-known Johnson-Lindenstrauss Lemma [1, 72], a random projection matrix $\boldsymbol{Q} \in \mathbb{R}^{s\times n}$ is applied to reduce the dimension of the data from n to $s = \mathcal{O}(\log n)$, while "nearly" preserve the pairwise DSD distance between nodes in the l_2 norm with high probability. This approach yields the approximate diffusion state $\widetilde{\boldsymbol{X}} := \boldsymbol{Q}\boldsymbol{D}^{\frac{1}{2}}\boldsymbol{N}^{\dagger}\boldsymbol{D}^{-\frac{1}{2}} \in \mathbb{R}^{s\times n}$. Each column of $\widetilde{\boldsymbol{X}}$ represents the coordinates of each vertex in a vector space of dimension $s = \mathcal{O}(\log n) \ll n$.

Based on the approximate diffusion state $\widetilde{X}$, the approximate DSD is defined as,

$$\widetilde{\mathrm{DSD}}(i,j) = \|\tilde{\boldsymbol{X}}(\boldsymbol{\psi}_i - \boldsymbol{\psi}_j)\|_p. \tag{1.14}$$

The following theorem states the quality of this approximation of DSD.

Theorem 1.2.3 ([90]) *Given $\epsilon, \gamma > 0$, let $s \geq \frac{4+2\gamma}{\epsilon^2-\epsilon^3}\log n$. Then, with probability at least $1 - n^{-\gamma}$, for all pair $0 \leq i, j \leq n$,*

$$\sqrt{1-\epsilon}\,\mathrm{DSD}(i,j) \leq \widetilde{\mathrm{DSD}}(i,j) \leq \sqrt{1+\epsilon}\,\mathrm{DSD}(i,j).$$

A different approximation of DSD was introduced in [48]. This spectral dimensional reduction approach projects data into a small number of coordinates, such that Euclidean distances in the embedded space approximate DSD in the original space. As pointed out in [48], it allows for the fast, faithful computation of the DSD by considering only the top eigenvectors of the underlying random walk. In addition, it denoises the DSD by truncating the eigenexpansion and discarding high-frequency eigenvectors that are typically corrupted by noise in the finite sample setting. Finally, it yields an interpretation of the DSD in terms of the principal eigenvectors of the random walk graph Laplacian.

1.2.2 Graph Learning Problem

Data analysis tasks, such as regression, classification, clustering, and visualization, naturally arise in various application domains. The problem size of these tasks also increases drastically as a result of the abundant availability of data sets. The study of machine learning can be divided into three main sub-fields [4]:

- **Unsupervised learning:** The training set consists of unlabelled data features, $\{x_1, x_2, \cdots, x_n\}$. Typical learning tasks include clustering (to find clusters or groupings of input), outlier detection (to identify data that is different from the other input), or dimension reduction.

- **Supervised learning:** The training set consists of a set of pairs of data features, x_i and their labels, y_i, denoted as $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$. The common goal of supervised learning is to predict the label, y, for a new input feature, x.

- **Reinforcement learning:** The system repeatedly interacts with the environment x by performing one, or a sequence of, actions, a, and receiving a reward r. Reinforcement learning aims to learn the actions that maximize the rewards.

In supervised learning, labeled data are used to train the regressor or classifier. However, obtaining the labeled data is often difficult, expensive, or time-consuming [142]. In many cases, a substantial effort from an experienced human annotator or expert input is required (for example, writing a transcript of speech utterance at the phonetic level for speech recognition or deciding whether a tumor is malignant or not based on the biopsy result). In contrast, unlabeled data can often be relatively easy to collect and comes in large quantities. Therefore, semi-supervised learning is developed where the training set typically consists of a relatively small labeled dataset $\{(x_1, y_1), (x_2, y_2), \cdots, (x_m, y_m)\}$ and a large unlabeled dataset $\{x_{m+1}, x_{m+2}, \cdots, x_{m+n}\}$. Semi-supervised learning harnesses the additional information present in unlabeled data to improve the learning results that only make

use of labeled data. For example, in a classification problem, the geometric or topological properties of unlabelled data provide valuable insights about where to place decision boundaries [60]. Semi-supervised learning achieves higher accuracy in machine learning tasks with less annotating effort [142] and therefore becomes a rapidly evolving field of both theoretic and practical interests. We refer to [121] for a comprehensive review of graph-based semi-supervised learning.

In graph-based semi-supervised learning, one of the most widely used methods is Laplace Learning proposed in 2003 in [143]. This is also known as label propagation on a graph, where a source node's label propagates to its neighboring nodes based on their proximity. The success of Graph Laplacian regularization is achieved because labels are propagated in such a way as to be consistent with the graph structure [136]. Other variants of Laplace learning and manifold learning were developed afterward [12, 132, 139, 142]. However, in scenarios where the semi-supervised graph learning tasks have very low label rates, Laplace learning often fails to propagate the label values well and tends to give very poor classification results [55, 100, 112]. Recent work has suggested different approaches to address this issue. Approaches like higher order Laplacian regularization [141] and spectral cutoffs [11] are proved to be useful. Another type of approach aims to re-weight the graph more heavily near the labeled nodes to give them wider influence when the labeling rate is very low. To do so, [119] proposed the Weighted Nonlocal Laplacian (WNLL) to amplify the weights of edges incident from the labeled nodes that improved upon the Laplace learning at moderately low label rates. Later, [35] presented the Properly Weighted Laplacian to re-weight the graph such that the learning is well-posed at arbitrarily low label rates.

In addition, [140] first generalized the graph Laplacian-based regularizers of [143] to the graph p-Laplacian regularizers. Graph-based p-Laplacian regularization has thereafter been applied for semi-supervised learning and image processing [56–58]. In [55] p-Laplace learning was applied to graph learning tasks with few labels. Several follow-up works, [32, 33, 112, 120], studied its variations and the properties of p-Laplace learning like the asymptotic behavior. Despite its improved performance

in label propagation and classification, p-Laplace learning is more computationally expensive, since its loss function is nonlinear. In section 1.2.2.2, we discuss an efficient and scalable algorithm to solve the variational p-Laplace equations. In a different effort to overcome the exact nature of the degeneracy that appeared in Laplace learning, [34] proposed the innovative Poisson learning algorithm that delivers good classification results even at extremely low label rates. We discuss the details of Laplace learning, Poisson learning, and p-Laplace learning in the rest of this section, and show that solving a linear or nonlinear graph Laplacian system efficiently is the key to these graph learning algorithms.

The Graph-based Semi-supervised Learning Problem: On a connected, undirected, and weighted graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{\omega})$, let $\mathcal{V} = \{x_1, x_2, \cdots, x_{n+m}\}$ denote the vertices of the graph, $\mathcal{E}$ the set of edges, and $\boldsymbol{\omega} = \{\omega_{ij}\}_{x_i, x_j \in \mathcal{V}}$ the set of non-negative edge weights. Assume each node is embedded in $\mathbb{R}^{n+m}$. Then, ω_{ij} measures how similar vertices x_i and x_j are. When x_i is similar to x_j, $\omega_{ij} \approx 1$, while $\omega \approx 0$ when x_i and x_j are dissimilar. A k-class classification, $k \geq 2$, is the problem of classifying data instances into one of the k different classes. We use the standard basis vector $\boldsymbol{e}_i \in \mathbb{R}^k$ to represent the i-th class for a k-class classification problem. We assume the last m nodes, $x_{n+1}, x_{n+2}, \cdots, x_{n+m}$ are labelled by $\boldsymbol{y}_1, \boldsymbol{y}_2, \cdots, \boldsymbol{y}_m \in \{\boldsymbol{e}_1, \boldsymbol{e}_2, \cdots, \boldsymbol{e}_k\}$. These labeled nodes form the observation set $\mathcal{B} = \{x_{n+1}, x_{n+2}, \cdots, x_{n+m}\} \subset \mathcal{V}$. A graph-based semi-supervised learning algorithm seeks to extend the labels from the observation set $\mathcal{B}$ to the rest of the vertices $\mathcal{V} \backslash \mathcal{B}$ in the graph in a meaningful way.

1.2.2.1 Laplace Learning and Poisson Learning

In 2003, Zhu et al. proposed the famous Laplace learning in [143] that extends the labels by solving the following:

$$\begin{aligned} &\boldsymbol{L}\boldsymbol{u}(x_i) = \boldsymbol{0}, \quad \text{if } 1 \leq i \leq n, \\ &\boldsymbol{u}(x_i) = \boldsymbol{y}_{i-n}, \quad \text{if } n+1 \leq i \leq n+m, \end{aligned} \tag{1.16}$$

where $\boldsymbol{L} \in \mathbb{R}^{(n+m)\times(n+m)}$ is the unnormalized graph Laplacian of $\mathcal{G}$, $\boldsymbol{u} : \mathcal{V} \mapsto \mathbb{R}^k$ is the labeling function, and $\boldsymbol{u}(\cdot) = [u_1(\cdot), u_2(\cdot), \cdots, u_k(\cdot)] \in \mathbb{R}^{(n+m)\times k}$. Then, the predicted label $\ell(x_i)$ for vertex x_i is assigned as the largest component of $\boldsymbol{u}(x_i)$,

$$\ell(x_i) = \underset{j\in\{1,\cdots,k\}}{\operatorname{argmax}}\{u_j(x_i)\}. \tag{1.17}$$

Graph Laplacian regularization enforces the learned labels to be consistent with the graph structure. Since two nodes that are close to each other are more likely to share the same labels, the learned labels should be smooth in the dense region of the graph. [40] also suggests that this semi-supervised *smoothness assumption* helps to moderate the a priori ill-posedness of the label-extending problem (there are infinitely many solutions to extend the labels). To enforce the smoothness assumption, a common approach is to define a regularizer, $J(\boldsymbol{u})$, to measure the smoothness of the labeling function $\boldsymbol{u}$. The regularizer, $J(\boldsymbol{u})$, is minimized subject to either hard label constraints, such that $\boldsymbol{u}(x)$ matches the known label of node x, or a soft penalty constraint. More details on the smoothness regularizers are discussed in section 1.2.2.2. We note that the Laplace equation, (1.16), can be solved by repeatedly replacing $\boldsymbol{u}(x_i)$ with the weighted average of its neighbors [142], which can be treated as a dynamical propagation of labels.

In the cases where the label rate is very low, Laplace learning generates solutions that develop localized spikes near the labeled points and are almost constant elsewhere, which fails to propagate the labels well. To address this issue, Calder et al proposed in [34] to replace (1.16) by Poisson Learning. Let $\overline{\boldsymbol{y}} = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{y}_i$ be the average label vector, and define,

$$\psi_{ij} = \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases}$$

Poisson learning extends the labels by computing the solution of,

$$\begin{aligned} &\boldsymbol{L}\boldsymbol{u}(x_i) = \sum_{j=1}^{m} (\boldsymbol{y}_j - \overline{\boldsymbol{y}})\psi_{(i-n)j}, \quad \text{for } i = 1, \cdots, n+m, \\ &\sum_{i=1}^{n+m} \delta_i \boldsymbol{u}(x_i) = 0, \end{aligned} \tag{1.19}$$

where $\delta_i = \sum_{j=1}^{n+m} \omega_{ij}$ is the weighted degree of node x_i. Once we learn the label function $\boldsymbol{u}$, the label decision is computed as in (1.17). In addition, the following modified label decision can help to address the unbalanced label class sizes, or a discrepancy between the balancing of training and testing dateset [34]:

$$\ell(x_i) = \underset{j \in \{1, \cdots, k\}}{\operatorname{argmax}} \{ s_j u_j(x_i) \}. \tag{1.20}$$

Here $s_j = b_j / (\overline{\boldsymbol{y}} \cdot \boldsymbol{e}_j)$ and b_j is the fraction of data belonging to label class j.

As pointed out in [34], in Laplace learning, the labels are imposed as boundary conditions in the Laplace equation (1.16), while in Poisson learning, the labels appear as a source term in the graph Poisson equation. Laplace learning works very efficiently in practice for graph-based semi-supervised learning problems with a moderate amount of labeled data, and Poisson learning is more powerful at very low label rates. We note that solving for either Laplace learning (1.16) or Poisson learning (1.19) involves solving a linear system with the graph Laplacian. We also refer to [34] for an iterative algorithm and implementation for Laplace and Poisson learning.

1.2.2.2 p-Laplace Learning

In this section, we focus on the analysis and algorithms for p-Laplace learning. In practice, when solving for the labeling function $\boldsymbol{u} : \mathcal{V} \mapsto \mathbb{R}^k$, the equations are separable among the coordinates of $\mathbb{R}^k$, and the problem reduces to solving for k functions $u_i : \mathcal{V} \mapsto \mathbb{R}$, one for each class $i = 1, \cdots, k$, which is usually referred to as the "one-vs-rest" approach in machine learning [60]. In this section, we follow the formulation in [60] and, without loss of generality, focus on the case where $k = 1$ for the analysis

for p-Laplace learning, and use the notation u for labeling function instead of $\boldsymbol{u}$. In addition, since our interest is in learning tasks with very few labels, we enforce the hard constraint that requires the learned labels to match the known labels exactly, but all the methods discussed in this section can be generalized directly, or with slight modifications, to problems with soft constraints (measuring the difference between learned labels and known labels with a penalty term).

We give a discussion on the standard Laplace operator before proceeding to the property of p-Laplace operator. The standard graph Laplacian $\boldsymbol{L} = \Delta_2^G(\cdot)$ defined in Equation (1.7) is a linear operator since $\boldsymbol{G}$ and $\boldsymbol{K}$ are linear. The operator $\Delta_2^G(\cdot) : \mathscr{V} \mapsto \mathscr{V}$ is also self-adjoint, since for any $\boldsymbol{v}_1, \boldsymbol{v}_2 \in \mathscr{V}$,

$$\begin{aligned}(\Delta_2^G(\boldsymbol{v}_1), \boldsymbol{v}_2) &= (\boldsymbol{G}^\top \boldsymbol{K} \boldsymbol{G} \boldsymbol{v}_1, \boldsymbol{v}_2) = (\boldsymbol{K}\boldsymbol{G}\boldsymbol{v}_1, \boldsymbol{G}\boldsymbol{v}_2) = (\boldsymbol{G}\boldsymbol{v}_1, \boldsymbol{K}\boldsymbol{G}\boldsymbol{v}_2) \\ &= (\boldsymbol{v}_1, \boldsymbol{G}^\top \boldsymbol{K} \boldsymbol{G} \boldsymbol{v}_2) = (\boldsymbol{v}_1, \Delta_2^G(\boldsymbol{v}_2)).\end{aligned}$$

The regularier $J_2(u)$ for Laplace learning is defined as,

$$J_2(u) \coloneqq \frac{1}{4} \sum_{i=1}^{n+m} \sum_{j=1}^{n+m} \omega_{ij} (u(x_i) - u(x_j))^2 = \frac{1}{4} (\boldsymbol{u}, \Delta_2^G(\boldsymbol{u})), \tag{1.21}$$

for $\boldsymbol{u} = [u(x_1), u(x_2), \cdots, u(x_{n+m})]$.

In Laplace learning, the learned labeling function u minimizes the functional $J_2(\cdot)$ and forces similar data points in the dense regions of the graph to have similar labels. Such a function u is called graph harmonic and it solves the graph 2-Laplace equation $\Delta_2^G u = 0$, where,

$$\Delta_2^G u(x_i) \coloneqq \sum_{j=1}^{n+m} \omega_{ij} (u(x_i) - u(x_j)).$$

Degeneracy: In the problems with very few labels, the sequence of learned functions obtained by Laplace learning does not continuously attain the labeled data in the continuum limit. In fact, the learned function becomes nearly constant except for sharp spikes near the labeled data, see Figure 1.2 (a) from [60] for an illustration

of the learned function by Laplace learning with 2 labeled and 10^5 unlabelled data points. Switching to p-Laplace learning helps to alleviate this issue.

We now generalize the standard Laplace operator $\Delta_2^G(\cdot)$ as the p-Laplace operator $\Delta_p^G(\cdot) : \mathcal{V} \mapsto \mathcal{V}$. For any $\boldsymbol{v} \in \mathcal{V}$,

$$\Delta_p^G(\boldsymbol{v}) := \boldsymbol{G}^\top \boldsymbol{K}(\|\boldsymbol{G}\boldsymbol{v}\|^{p-2}\boldsymbol{G}\boldsymbol{v}).$$

Note that we recover the standard graph Laplacians for $p = 2$. The p-Laplace operator is nonlinear, that is, $\Delta_p^G(a\boldsymbol{v}) \neq a\Delta_p^G(\boldsymbol{v})$. Then the smoothness functional corresponding to the ℓ_p-based Laplacian regularization is:

$$J_p(u) := \frac{1}{2p}\sum_{i=1}^{n+m}\sum_{j=1}^{n+m}\omega_{ij}|u(x_i) - u(x_j)|^p = \frac{1}{2p}(\boldsymbol{u}, \Delta_p^G(\boldsymbol{u})).$$

If $p > 2$, $J_p(u)$ places an even heavier penalty on large gradients $|u(x_i) - u(x_j)|$, and as a result, discourages the solution from developing sharp spikes. The u that minimizes $J_p(\cdot)$ satisfies the graph p-Laplace equation $\Delta_p^G u = 0$, where

$$\Delta_p^G u(x_i) := \sum_{j=1}^{n+m}\omega_{ij}|u(x_j) - u(x_i)|^{p-2}(u(x_j) - u(x_i)). \tag{1.22}$$

The ℓ_p-based Laplacian regularized learning problem is:

$$\begin{aligned}&\min_{u:\mathcal{V}\mapsto\mathbb{R}} J_p(u),\\ &\text{subject to: } u(x) = g(x) \quad \text{if } x \in \mathcal{B}.\end{aligned}$$

The minimizer, $u : \mathcal{V} \mapsto \mathbb{R}$, of the above constraint minimization problem satisfies:

$$\begin{cases} -\Delta_p^G u(x) = 0 & \text{if } x \in \mathcal{V}\backslash\mathcal{B}, \\ u(x) = g(x) & \text{if } x \in \mathcal{B}. \end{cases} \tag{1.23}$$

As p increases, the learned function transitions more smoothly between labeled and unlabeled points. Revisiting the example in Figure 1.2 from [60], we see that

as p increases, the surface of the learned function becomes smoother, while for $p = 2$ the surface is nearly constant except the two spikes near the labeled nodes.

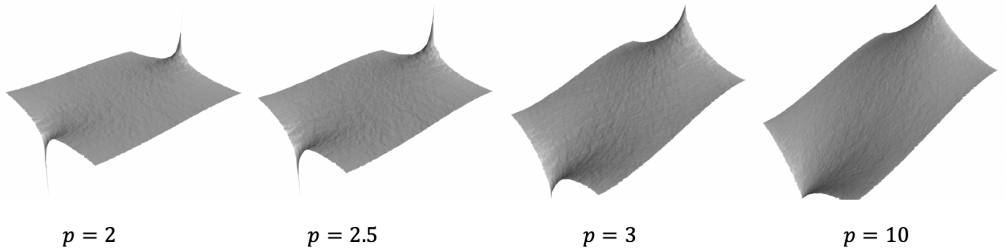


Figure 1.2: Learned functions based on p-Laplacian regularization using different p value with two labeled and 10^5 unlabeled data points on $[0,1]^2$. This example is from [60].

To include more general learning scenarios, for a source function $f : \mathcal{V}\backslash\mathcal{B} \mapsto \mathbb{R}$, we generalize the definition of $J_p(u)$ subject to label constraints to be:

$$\begin{aligned} &J_p(u) := \frac{1}{2p}\sum_{i=1}^{n+m}\sum_{j=1}^{n+m}\omega_{ij}|u(x_i)-u(x_j)|^p + \sum_{i=1}^{n} f(x_i)u(x_i), \\ &\text{subject to: } u(x) = g(x) \quad \text{if } x \in \mathcal{B}. \end{aligned} \tag{1.24}$$

The unique minimizer $u : \mathcal{X} \mapsto \mathbb{R}$ of (1.24) satisfies:

$$\begin{cases} -\Delta_p^G u(x) = f(x) & \text{if } x \in \mathcal{V}\backslash\mathcal{B}, \\ u(x) = g(x) & \text{if } x \in \mathcal{B}. \end{cases} \tag{1.25}$$

Newton's Method for p-Laplace Learning

We would like to use Newton's Method to minimize the objective function in Equation (1.24) with the constraints, since $J_p(\cdot)$ is smooth and convex.

Denote $u_i = u(x_i)$, $f_i = f(x_i)$, for $i = 1,\cdots,n$, and $g_j = g(x_{n+j})$, for $j = 1,\cdots,m$. We write the minimization problem subject to boundary constraints, encoding in the known labels, as,

$$
\begin{aligned}
J_p(u) &= \frac{1}{2p}\left(\sum_{i=1}^{n+m}\sum_{j=1}^{n+m} w_{ij}|u_i-u_j|^p\right) - \sum_{i=1}^{n} f_i u_i \\
&= \frac{1}{2p}\Bigg(\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}|u_i-u_j|^p + \sum_{i=1}^{n}\sum_{j=1}^{m} w_{i,j+n}|u_i-g_j|^p + \sum_{i=1}^{m}\sum_{j=1}^{n} w_{i+n,j}|g_i-u_j|^p + \\
&\quad \sum_{i=1}^{m}\sum_{j=1}^{m} w_{i+n,j+n}|g_i-g_j|^p\Bigg) + \sum_{i=1}^{n} f_i u_i \\
&= \frac{1}{p}\left(\sum_{i=1}^{n}\sum_{j=i+1}^{n} w_{ij}|u_i-u_j|^p + \sum_{i=1}^{n}\sum_{j=1}^{m} w_{i,j+n}|u_i-g_j|^p\right) + \sum_{i=1}^{n} f_i u_i + \text{constant}.
\end{aligned}
\tag{1.26}
$$

Now the minimization is computed with respect to the interior nodes $\boldsymbol{u}_{\text{int}} = (u_1, \cdots, u_n) \in \mathbb{R}^n$. For the simplicity of notation, we still denote it as $\boldsymbol{u} = \boldsymbol{u}_{\text{int}}$. To minimize

$$
J_p(\boldsymbol{u}) = \frac{1}{p}\left(\sum_{i=1}^{n}\sum_{j=i+1}^{n} w_{ij}|u_i-u_j|^p + \sum_{i=1}^{n}\sum_{j=1}^{m} w_{i,j+n}|u_i-g_j|^p\right) + \sum_{i=1}^{n} f_i u_i, \tag{1.27}
$$

the Newton's iteration is written out explicitly as,

$$
\boldsymbol{u}^{k+1} = \boldsymbol{u}^k - \left[\nabla^2 J_p(\boldsymbol{u}^k)\right]^{-1} \nabla J_p(\boldsymbol{u}^k),
$$

where $\nabla^2 J_p(\boldsymbol{u}) \in \mathbb{R}^{n\times n}$ is the Hessian matrix of $J_p(\cdot)$ evaluated at $\boldsymbol{u}$, and $\nabla J_p(\boldsymbol{u}) \in \mathbb{R}^n$ is the gradient matrix of $J_p(\cdot)$ evaluated at $\boldsymbol{u}$.

We denote:

$$
\begin{aligned}
&a_{ij}(\boldsymbol{u}) = w_{ij}|u_i-u_j|^{p-2},\ i=1,\cdots,n, j=1,\cdots,n, \\
&b_{ij}(\boldsymbol{u}) = w_{i,j+n}|u_i-g_j|^{p-2},\ i=1,\cdots,n, j=1,\cdots,m, \\
&d_i(\boldsymbol{u}) = \sum_{j=1}^{n} a_{ij}(\boldsymbol{u}) + \sum_{j=1}^{n} b_{ij}(\boldsymbol{u}),\ i=1,\cdots,n, \\
&\boldsymbol{A}(\boldsymbol{u}) \in \mathbb{R}^{n\times n}, \quad [\boldsymbol{A}(\boldsymbol{u})]_{i,j} = a_{ij}, \\
&\boldsymbol{B}(\boldsymbol{u}) \in \mathbb{R}^{n\times m}, \quad [\boldsymbol{B}(\boldsymbol{u})]_{i,j} = b_{ij} \\
&\boldsymbol{D}(\boldsymbol{u}) = \text{diag}(d_i(\boldsymbol{u})) \in \mathbb{R}^{n\times n}, \quad \boldsymbol{\mathcal{L}}(\boldsymbol{u}) = \boldsymbol{D}(\boldsymbol{u}) - \boldsymbol{A}(\boldsymbol{u}) \in \mathbb{R}^{n\times n}, \\
&\boldsymbol{f} = (f_1, \cdots, f_n) \in \mathbb{R}^n, \quad \boldsymbol{g} = (g_1, \cdots, g_m) \in \mathbb{R}^m.
\end{aligned}
\tag{1.29}
$$

Then, $\nabla^2 J_p(\boldsymbol{u})$ and $\nabla J_p(\boldsymbol{u})$ can be written as:

$$\nabla J_p(\boldsymbol{u}) = \boldsymbol{\mathcal{L}}(\boldsymbol{u})\boldsymbol{u} - \boldsymbol{B}(\boldsymbol{u})\boldsymbol{g} + \boldsymbol{f},$$
$$\nabla^2 J_p(\boldsymbol{u}) = (p-1)\boldsymbol{\mathcal{L}}(\boldsymbol{u}).$$

Therefore, Newton's iteration becomes:

$$\begin{aligned}\boldsymbol{u}^{k+1} &= \boldsymbol{u}^k - \left[\nabla^2 J_p(\boldsymbol{u}^k)\right]^{-1} \nabla J_p(\boldsymbol{u}^k) \\ &= \boldsymbol{u}^k - \left[(p-1)\boldsymbol{\mathcal{L}}(\boldsymbol{u}^k)\right]^{-1}\left(\boldsymbol{\mathcal{L}}(\boldsymbol{u}^k)\boldsymbol{u}^k - \boldsymbol{B}(\boldsymbol{u})\boldsymbol{g} + \boldsymbol{f}\right) \\ &= \frac{p-2}{p-1}\boldsymbol{u}^k + \frac{1}{p-1}\boldsymbol{\mathcal{L}}(\boldsymbol{u}^k)^{-1}\left[\boldsymbol{B}(\boldsymbol{u}^k)\boldsymbol{g} - \boldsymbol{f}\right]\end{aligned} \tag{1.31}$$

We point out the need to invert a Graph Laplacian $\boldsymbol{\mathcal{L}}(\boldsymbol{u}^k)$ in each Newton iteration, which is equivalent to solving a graph Laplacian system with the identity matrix on the right-hand side. In the next section, we discuss ways to solve systems of equations with graph Laplacians efficiently. Also, in Chapter 4, we present a nested iteration algorithm to take advantage of the multilevel scheme and further reduce the computational cost of p-Laplace learning.

1.3 Solving Graph Laplacian Systems with Algebraic Multigrid

A system of linear equations with graph Laplacian $\boldsymbol{L}$ of a simple connected weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{\omega})$ takes the general form of,

$$\boldsymbol{L}\boldsymbol{x} = \boldsymbol{f}. \tag{1.32}$$

Existing solvers of linear systems generally fall into two categories: direct methods and iterative methods. The direct methods, prototyped by Gaussian elimination and its variants (Cholesky, LU, QR, etc.), determine a solution exactly (up to machine precision) in a finite number of steps if the system is invertible. See [52] for a survey

of the theory and practice of direct methods for solving linear systems and least-squares problems. We note that $\boldsymbol{L}$ is singular, and its null space is spanned by the all-one vector $\mathbf{1}$. In this case, the iterative methods are more suitable and robust to solve large-scale graph Laplacian linear systems.

Iterative methods, as represented by the Jacobi and Gauss–Seidel iterations, begin with an initial guess and provide a sequence of successively improved approximations to the exact solution. One fundamental family of iterative methods is the Krylov subspace methods [5]. Some of the widely used Krylov subspace methods include the Generalized Minimal Residual Method (GMRES) [116] for non-symmetric linear systems and the Conjugate Gradient method (CG) [117] for symmetric positive definite linear systems. We say that a computation is ill-conditioned or not well-posed to solve Equation (1.32) if "small" changes in $\boldsymbol{f}$ cause "big" changes in the solution $\boldsymbol{x}$, and a condition number for $\boldsymbol{L}$ serves as a measure of ill-conditioning [71]. The condition number, $\kappa(\cdot)$, for a matrix $\boldsymbol{L}$ and a choice of matrix norm $\|\cdot\|$ is defined to be,

$$\kappa(\boldsymbol{L}) \coloneqq \|\boldsymbol{L}^{-1}\|\|\boldsymbol{L}\|.$$

When the condition number $\kappa(\boldsymbol{L})$ is not too large, the system (1.32) is well-conditioned [71]. To further accelerate iterative methods when solving linear systems with large condition numbers, preconditioners can be used. In Equation (1.32), if the conditioning number $\kappa(\boldsymbol{L})$ for $\boldsymbol{L}$ is large, we can precondition the system with a matrix $\boldsymbol{M}$ such that the linear system,

$$\boldsymbol{M}\boldsymbol{L}\boldsymbol{x} = \boldsymbol{M}\boldsymbol{f},$$

is easier to solve. That is, the new system with $\boldsymbol{M}\boldsymbol{L}$ should have a smaller condition number, which leads us to choose the $\boldsymbol{M}$ that approximates the inverse or the Moore-Penrose pseudoinverse of $\boldsymbol{L}$. See this survey [13] for more technical details on preconditioning.

Algebraic multigrid (AMG) methods, originally proposed in [22], are often applied to solve linear systems (see also [134] and the references there for a recent

survey on the AMG methods). In practice, AMG methods achieve optimal computational complexity for many applications, including solving the linear systems with weighted graph Laplacians [17, 24, 50, 78, 81, 92, 103].

1.3.1 Algebraic Multigrid Method

A typical multigrid method traverses a hierarchy of spaces (grids) of different dimensions (grid sizes) and solves the corresponding linear system at different resolutions. Multilevel methods cleverly use solutions on coarser grids to improve the solution on the fine grid and avoid the need to directly solve the system on the fine grid, which is more computationally challenging. We outline an example of the two-level method in Algorithm 1 below and refer the readers to [29] for a more comprehensive introduction to the multigrid method.

In Algorithm 1, we solve the fine-level linear system $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{f}$. Here, $\boldsymbol{S}$ is a relaxation method that is usually chosen to be the Jacobi or the Gauss-Seidel iteration. We use $\boldsymbol{R}$ to denote the restriction operator that maps from the fine grid to the coarse grid, while $\boldsymbol{R}^\top$ is the prolongation operator that maps from the coarse to the fine grid. The coarse matrix $\boldsymbol{L}_c = \boldsymbol{R}\boldsymbol{L}\boldsymbol{R}^\top$.

Algorithm 1 A Two-level Scheme

1: Relaxation on fine grid: $\boldsymbol{x} = \boldsymbol{x} + \boldsymbol{S}(\boldsymbol{f} - \boldsymbol{L}\boldsymbol{x})$.

2: Restrict residual to coarse grid: $\boldsymbol{r}_c = \boldsymbol{R}(\boldsymbol{f} - \boldsymbol{L}\boldsymbol{x})$.

3: Compute coarse grid correction: $\boldsymbol{e}_c = \boldsymbol{L}_c^{-1}\boldsymbol{r}_c$.

4: Prolongate correction to fine grid: $\boldsymbol{x} = \boldsymbol{x} + \boldsymbol{R}^\top \boldsymbol{e}_c$.

5: Relaxation on fine grid: $\boldsymbol{x} = \boldsymbol{x} + \boldsymbol{S}(\boldsymbol{f} - \boldsymbol{L}\boldsymbol{x})$.

An efficient AMG method should damp the algebraic high-frequency error using relaxation/smoothers and eliminate the algebraically smooth error via a coarse space (grid) correction [29]. The latter requires the "smooth" error obtained on the fine grids to be accurately approximated on the coarse spaces. Many different coarsening strategies have been developed based on good estimations of the error, for example, classical AMG [22, 27], smoothed aggregation AMG [25, 26, 28], bootstrap AMG

[20, 21], and unsmoothed aggregation AMG [23, 24, 78, 92, 102, 108, 126]. Thus, an efficient, reliable, and computable estimate of the error a posteriori, during AMG iterations, is needed for developing robust AMG methods.

1.4 Outline

Chapter 1 reviews the background of the graph representations, applications of graph Laplacian systems in computational biology and machine learning problems, and AMG algorithms to solve graph Laplacian systems. The rest of the thesis is organized as follows. In Chapter 2, we derive and analyze a novel a posteriori error estimator that is essential in the effectiveness of AMG methods. Chapter 3 and Chapter 4 present two applications that involve solving the linear systems of graph Laplacians. In Chapter 3, we employ a graph diffusion based distance metric DSD to study the intricate biological networks. A spectral dimension reduction technique is also implemented to help reduce the computational complexity while approximating the DSD with high fidelity. In Chapter 4, we analyze and compare several graph-learning algorithms in different learning tasks.

Chapter 2

Solving Graph Laplacians by Algebraic Multigrid with a Posteriori Error Estimates

In the previous chapter, we discussed how graphs and large linear systems with graph Laplacians are used in a wide range of applications, and therefore, why it is important to develop efficient and robust methods for solving graph Laplacian systems. In practice, the AMG method achieves optimal computational complexity for many applications, including solving linear systems with weighted graph Laplacians [17, 24, 50, 78, 81, 92, 103]. This chapter is dedicated to introducing a novel a posteriori error estimator which aids AMG solver for linear systems with graph Laplacians.

Generally, a posteriori estimators provide a computable estimation for the true error locally. Our approach borrows several ideas from the finite-element (FE) literature (equilibrated error estimators [2, 6, 74, 127] and functional a posteriori error estimators [53, 107, 111, 125]). In [135], the authors derived a posteriori error estimator for solving graph Laplacian linear systems for the first time based on the functional a posteriori error estimation framework. Such a technique was used to predict the error of approximation from coarse grids for multilevel unsmoothed aggregation AMG and the estimator is computed by solving a perturbed global optimization problem (discussed in more detail in Section 2.1). Such an approach provides an accurate error estimator. However, it could be computationally expensive, which affects the effectiveness of the resulting adaptive AMG method. In this work, we propose a novel a posteriori error estimator and an efficient algorithm to reduce the computational cost, which could further be used to construct the efficient multilevel hierarchy for adaptive AMG. Roughly speaking, this is achieved by taking advantage of the Helmholtz decomposition [79] on the graph computationally, which splits the

error into a divergence-free component and a curl-free component. The rationale of the proposed algorithm for computing the approximation to the true error has two main steps:

1. Solve a linear system on a spanning tree (defined in Section 2.1) of the graph to get the curl-free component, or equivalently, the grad component of the error.

2. Approximately solve a minimization problem in the cycle space to obtain the div-free component of the Helmholtz decomposition of the error.

The first step can be done in linear time with Gaussian elimination using lexicographical ordering as shown in [113, 128]. Solving exactly the minimization in the cycle space of the graph, though, is computationally expensive as it is equivalent to solving a constrained minimization problem, which is as difficult as the original linear system (often even more difficult). Our algorithm solves the minimization approximately by applying several steps of a relaxation scheme, the one-level Schwarz method [41, 124]. This crucial improvement reduces the computational cost and gives an accurate a posteriori estimates of the true error in nearly optimal time for sparse graphs, which is further verified by our numerical experiments. Clearly, such an error estimator can be incorporated to construct multilevel hierarchies since it provides accurate estimate of the true error, which is important for constructing coarse levels adaptively. The corresponding adaptive coarsening scheme will preserve the smooth error accurately on the coarse levels and ensure the robustness of the resulting adaptive AMG method.

The rest of this chapter is organized as follows. In Section 2.1 we introduce the essential concepts to follow the derivation of such an error estimates, along with some previous results in [135]. The main algorithm to compute a posterior error estimates is stated in Section 2.2. We present and analyze some numerical experiments in Section 2.3. Finally, in Section 2.5 we summarize the main contribution and list some future work.

2.1 Background on a Posteriori Error Estimates

In this section, we define the necessary notations in addition to the ones defined in Chapter 1, and recall some fundamental results for the computation of an a posteriori error estimator for solving graph Laplacians as presented in [135].

2.1.1 Spanning Tree and Cycle Space on Graphs

In addition to the vertex space $\mathscr{V}$ and edge space $\mathscr{W}$, another important space of a graph $\mathcal{G}$ is the so-called *cycle space* (see [16] for more details), denoted by $\mathscr{C}$, which is defined as,

$$\mathscr{C} := \left\{ \boldsymbol{c} \in \mathscr{W} \mid \boldsymbol{G}^\top \boldsymbol{c} = \boldsymbol{0} \right\}. \tag{2.1}$$

Each cycle on the graph $\mathcal{G}$ corresponds to an element $\boldsymbol{c}$ in the cycle space $\mathscr{C}$. To be more specific, if $\{i_1, i_2, \ldots, i_k, i_{k+1} = i_1\}$ is a cycle $\mathcal{G}$, that is,

$$i_j \in \mathcal{V}, \quad \text{and} \quad \{\max\{i_j, i_{j+1}\}, \min\{i_j, i_{j+1}\}\} \in \mathcal{E}, \quad j = 1, \ldots, k,$$

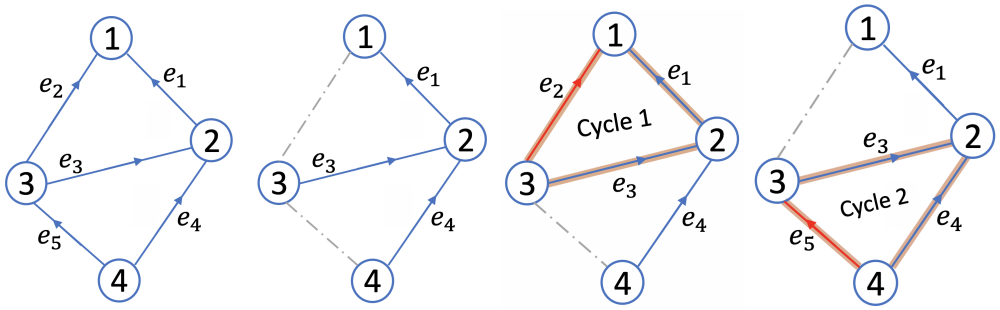
we define the components of $\boldsymbol{c} \in \mathscr{C}$ as follows:

$$(\boldsymbol{c})_e = \operatorname{sign}(i_j - i_{j+1}), \quad \text{if} \quad e = \{\max\{i_j, i_{j+1}\}, \min\{i_j, i_{j+1}\}\},$$

with $(\boldsymbol{c})_e$ extended as zero for all edges that are not on the cycle. Besides its definition (2.1), we can also characterize the cycle space $\mathscr{C}$ by its basis. As discussed in [73], the cycle space of a connected simple graph has dimension $m - n + 1$ (by simple graphs we mean the graphs that contains neither self-loops nor duplicate edges). Note there is more than one way to find the basis of the cycle space (see the survey paper [73]). For general graphs, a commonly used set of basis for the cycle space is the *basis of fundamental cycles*. Such basis is not unique, but each such basis is induced by a spanning tree. In a connected graph $\mathcal{G}$, with n vertices, a spanning tree is any subgraph of $\mathcal{G}$ that connects all the vertices and has no cycles or, equivalently, has exactly $(n-1)$ edges. To construct the basis of fundamental

cycles corresponding to a spanning tree $\mathcal{T} = (\mathcal{V}, \mathcal{E}_\mathcal{T}, \omega_\mathcal{T})$ of a graph $\mathcal{G}$, we proceed as follows. For each edge that does not belong to the tree, i.e., $e = \{i, j\} \in \mathcal{E} \backslash \mathcal{E}_\mathcal{T}$, we can find a cycle $\{i, j\} \cup p(i, j)$ where $p(i, j)$ is the path from vertex i to vertex j on the tree $\mathcal{T}$. Since the spanning tree $\mathcal{T}$ has exactly $(n-1)$ edges, there are $(m-n+1)$ such cycles. It can be shown that they are linearly independent [73] and, therefore, form a basis for the cycle space $\mathscr{C}$.

In 2.1, we give a simple example of the fundamental cycle basis. The tree in 2.1(b) is a spanning tree of the graph in 2.1(a). First, the edge e_2 is added back (see 2.1(c)) which results in the first cycle $\{2, 1, 3, 2\}$ consisting of edges e_1, e_2, and e_3. The vector representation of the cycle induced by adding back edge e_2 is given by $\boldsymbol{c}^{e_2} = [1, -1, 1, 0, 0]^\top$. Similarly, by adding edge e_5 back, we have the second cycle $\{2, 3, 4, 2\}$ formed by edges e_3, e_4 and e_5, which is represented by $\boldsymbol{c}^{e_5} = [0, 0, -1, -1, 1]^\top$. $\boldsymbol{c}^{e_2}$ and $\boldsymbol{c}^{e_5}$ form a cycle basis.



(a) original graph (b) spanning tree (c) adding edge e_2 to get cycle 1. (d) add edge e_5 to get cycle 2.

Figure 2.1: Fundamental cycle basis

2.1.2 Previous Results on A Posteriori Error Estimators

We are interested in solving the following linear system of graph Laplacians:

$$\boldsymbol{L}\boldsymbol{u} = \boldsymbol{f}, \tag{2.2}$$

by some iterative methods. Equation (2.2) simulates a rich spectrum of weighted graph Laplacians problems [70, 90, 98, 114]. For example, in computational physics,

the solution $\boldsymbol{u} \in \mathbb{R}^n$ models the potentials of an electrical flow where the resistance of each edge on the graph (electrical circuit) is the reciprocal of the edge weight, and $\boldsymbol{f} = [f_1, f_2, \cdots, f_n]^\top \in \mathbb{R}^n$ denotes the current supplied to each node i.

After k iterations, we get an approximated solution $\boldsymbol{u}^k$. If we can somehow construct the current error, $\boldsymbol{e}^k = \boldsymbol{u} - \boldsymbol{u}^k$, then the true solution is easily obtained by $\boldsymbol{u} = \boldsymbol{u}^k + \boldsymbol{e}^k$. In practice, the true error $\boldsymbol{e}^k$ is not computable because $\boldsymbol{u}$ is unknown, so alternatively we seek to find $\widetilde{\boldsymbol{e}}^k$, an accurate estimation of $\boldsymbol{e}^k$, and use $\widetilde{\boldsymbol{e}}^k$ to improve the current approximation. Furthermore, an accurate estimation of the error gives an insight into the performance of the iterative methods. For example, in AMG methods, such an estimation approximates the so-called "smooth error", which is responsible for the slow convergence of the AMG methods, and can be used to improve the AMG algorithm adaptively. This leads to adaptive AMG methods [25, 27, 50, 95, 101], which have been actively researched over the past two decades.

The a posteriori estimator presented here is motivated by the a posteriori error estimator developed in [135]. Thus, we recall the main results and algorithms presented in [135] and start with the following fundamental lemma which relates the error and computed approximate solution. The proof of the lemma is found in [135].

Lemma 2.1.1 *[135, Lemma 3.2] Let $\boldsymbol{u}$ be the solution to Equation (2.2). Then for arbitrary $\boldsymbol{\tau} \in \mathscr{W}$, the following inequality holds for all $\boldsymbol{v} \in \mathscr{V}$:*

$$\|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}} \leq \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}} + C_p^{-1}\|\boldsymbol{G}^\top\boldsymbol{\tau} - \boldsymbol{f}\|_{\mathscr{V}}. \tag{2.3}$$

where C_p is Poincaré's constant of the graph Laplacian $\boldsymbol{L}$.

For a fixed $\boldsymbol{v}$, denote the right-hand side of Equation (2.3) by:

$$\eta(\boldsymbol{\tau}) = \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}} + C_p^{-1}\|\boldsymbol{G}^\top\boldsymbol{\tau} - \boldsymbol{f}\|_{\mathscr{V}}.$$

This naturally provides a posteriori error estimator for estimating the error $\boldsymbol{u} - \boldsymbol{v}$ if $\boldsymbol{v}$ is an approximate solution, i.e., $\boldsymbol{v} = \boldsymbol{u}^k$. Moreover, by minimizing the right-hand

side of Equation (2.3) with respect to $\boldsymbol{\tau}$, we can obtain an accurate estimator. To solve the minimization problem efficiently, in [135], an upper bound $E(\beta, \boldsymbol{\tau})$ of $\eta(\boldsymbol{\tau})$ was introduced as follows,

$$\eta^2(\boldsymbol{\tau}) \leq E(\beta, \boldsymbol{\tau}),$$

where

$$E(\beta, \boldsymbol{\tau}) := (1+\beta)\|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}\|^2_{\boldsymbol{K}^{-1}} + (1+\frac{1}{\beta})C_p^{-1}\|\boldsymbol{G}^\top\boldsymbol{\tau} - \boldsymbol{f}\|^2_{\mathscr{V}'}.$$

And an accurate estimator can be obtained by computing $\min_{\beta,\boldsymbol{\tau}} E(\beta, \boldsymbol{\tau})$. In [135], an alternating process is applied to minimize $E(\beta, \boldsymbol{\tau})$ with respect to β (with the techniques proposed in [82]) and $\boldsymbol{\tau}$ iteratively, as summarized in Algorithm 2 (see [135] for details):

Algorithm 2 Alternating Process for Solving $\min_{\beta,\boldsymbol{\tau}} E(\beta, \boldsymbol{\tau})$

1: **procedure** $[\beta, \boldsymbol{\tau}]$ =MinimizeBound$(\beta^0, \boldsymbol{\tau}^0)$
2: **for** $k = 1, 2, \cdots,$ max_iter **do**
3: compute $\boldsymbol{\tau}^k = \text{argmin}_{\boldsymbol{\tau}} E(\beta^{k-1}, \boldsymbol{\tau})$.
4: compute $\beta^k = \text{argmin}_{\beta} E(\beta, \boldsymbol{\tau}^k)$.
5: **end for**
6: **end procedure**

Although the approach developed in [135] provides a reliable error estimator, the corresponding computational cost might be expensive due to the iterative minimization of $E(\beta, \boldsymbol{\tau})$ in step 3 and 4 in Algorithm 2. One iteration of step 4 can be as difficult as solving the original system, which makes this approach expensive computationally. In order to improve the accuracy of the a posteriori error estimator, and, more importantly, to improve the efficiency of computing it, we develop a novel technique for estimating the error based on Equation (2.3), which we present next.

2.2 Efficient Algorithm for Computing A Posteriori Error Estimator

As we have pointed out earlier in this Chapter, the derivation of the a posteriori error estimator is based on approximating the Helmholtz decomposition of the error.

This provides a tighter error bound than the one proposed in [135] and can be implemented efficiently.

2.2.1 Hypercircle Identity and Error Estimation on Graphs

Our design of an a posteriori error estimator is motivated by Equation (2.3). For a given $\boldsymbol{f} \in \mathscr{V}$, we define the space $\mathscr{W}(\boldsymbol{f}) = \left\{\boldsymbol{\tau} \in \mathscr{W} \mid \boldsymbol{G}^{\top}\boldsymbol{\tau} = \boldsymbol{f}\right\}$. If we choose $\boldsymbol{\tau} \in \mathscr{W}(\boldsymbol{f})$, then the second term on the right-hand side of Equation (2.3) vanishes and we only have the first term left. If we minimize this term with respect to $\boldsymbol{\tau} \in \mathscr{W}(\boldsymbol{f})$, we can immediately get an accurate estimation. We summarize this in the following theorem.

Theorem 2.2.1 *Let $\boldsymbol{u}$ be the solution to Equation* (2.2)*. Then for any $\boldsymbol{v} \in \mathscr{V}$, we have,*

$$\|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}} = \min_{\boldsymbol{\tau} \in \mathscr{W}(\boldsymbol{f})} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}. \tag{2.4}$$

To prove Theorem (2.2.1), we make use of the next lemma, also known as the *hypercyrcle identity* (see [110]for its proof):

Lemma 2.2.2 *Let $\boldsymbol{u}$ be the solution to* (2.2)*. Then for any $\boldsymbol{v} \in \mathscr{V}$ and any $\boldsymbol{\tau} \in \mathscr{W}(f)$ the following identity holds:*

$$\|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}}^2 + \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{u} - \boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}^2 = \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}^2.$$

Now we are ready to prove Theorem 2.2.1.

Proof: (Theorem 2.2.1) We first show $\|\boldsymbol{u}-\boldsymbol{v}\|_L \leq \min_{\boldsymbol{\tau} \in \mathscr{W}(\boldsymbol{f})} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v}-\boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}$. It follows from Lemma 2.2.2 that,

$$\|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}}^2 = \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}^2 - \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{u} - \boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}^2 \leq \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}^2.$$

Since the inequality holds for any $\boldsymbol{\tau} \in \mathscr{W}(\boldsymbol{f})$, we have:

$$\|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}}^2 \leq \min_{\boldsymbol{\tau} \in \mathscr{W}(\boldsymbol{f})} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}^2.$$

To show the other direction, note that,

$$\begin{aligned}\|\boldsymbol{u}-\boldsymbol{v}\|_{\boldsymbol{L}}^2 &= (\boldsymbol{L}(\boldsymbol{u}-\boldsymbol{v}),\boldsymbol{u}-\boldsymbol{v}) = (\boldsymbol{G}^\top\boldsymbol{K}\boldsymbol{G}(\boldsymbol{u}-\boldsymbol{v}),\boldsymbol{u}-\boldsymbol{v}) = (\boldsymbol{K}\boldsymbol{G}(\boldsymbol{u}-\boldsymbol{v}),\boldsymbol{G}(\boldsymbol{u}-\boldsymbol{v}))\\ &= (\boldsymbol{K}\boldsymbol{G}(\boldsymbol{u}-\boldsymbol{v}),\boldsymbol{K}\boldsymbol{G}(\boldsymbol{u}-\boldsymbol{v}))_{\boldsymbol{K}^{-1}} = \|\boldsymbol{K}\boldsymbol{G}(\boldsymbol{u}-\boldsymbol{v})\|_{\boldsymbol{K}^{-1}}^2 = \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v}-\boldsymbol{K}\boldsymbol{G}\boldsymbol{u}\|_{\boldsymbol{K}^{-1}}^2\\ &\geq \min_{\boldsymbol{\tau}\in\mathscr{W}(\boldsymbol{f})} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v}-\boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}^2.\end{aligned}$$

This completes the proof. □

From Theorem 2.2.1, we observe that,

$$\|\boldsymbol{u}-\boldsymbol{v}\|_{\boldsymbol{L}} \leq \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v}-\boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}},$$

for any $\boldsymbol{\tau}\in\mathscr{W}(\boldsymbol{f})$. This motivates us to define the following computable quantity,

$$\psi(\boldsymbol{\tau}) := \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v}-\boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}, \quad \forall\,\boldsymbol{\tau}\in\mathscr{W}(\boldsymbol{f}). \tag{2.5}$$

If $\boldsymbol{v}$ is the approximate solution to Equation (2.2), $\psi(\boldsymbol{\tau})$ gives an a posteriori estimator for the true error $\boldsymbol{u}-\boldsymbol{v}$ for any choice of $\boldsymbol{\tau}\in\mathscr{W}(\boldsymbol{f})$. If $\boldsymbol{\tau}^*$ is the minimizer of the right-hand side of Equation (2.4), then $\psi(\boldsymbol{\tau}^*) = \|\boldsymbol{u}-\boldsymbol{v}\|_{\boldsymbol{L}}$. Of course, computing the minimizer $\boldsymbol{\tau}^*$ exactly would be expensive. As an alternative, we propose a Schwarz method for approximating $\boldsymbol{\tau}^*$. As the numerical tests show, we obtain a reasonably good approximation $\boldsymbol{\tau}\in\mathscr{W}(\boldsymbol{f})$, $\boldsymbol{\tau}\approx\boldsymbol{\tau}^*$.

2.2.2 Efficient Evaluation of the Error Estimator

Our approach is to solve the minimization problem,

$$\min_{\boldsymbol{\tau}\in\mathscr{W}(\boldsymbol{f})} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v}-\boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}, \tag{2.6}$$

based on the Helmholtz decomposition of $\boldsymbol{\tau}$ on the graph, i.e., $\boldsymbol{\tau} = \boldsymbol{\tau}_f + \boldsymbol{\tau}_0^*$, where $\boldsymbol{\tau}_f\in\mathscr{W}(\boldsymbol{f})$ and $\boldsymbol{\tau}^*\in\mathscr{C}$ such that $(\boldsymbol{\tau}_f,\boldsymbol{\tau}_0^*) = 0$. Here $\boldsymbol{\tau}_f$ is curl-free and $\boldsymbol{\tau}_0^*$ is divergence-free. In particular, we first find a $\boldsymbol{\tau}_f\in\mathscr{W}(\boldsymbol{f})$ by solving a graph Laplacian on a spanning tree of the graph. Then, for a given $\boldsymbol{\tau}_f\in\mathscr{W}(\boldsymbol{f})$, the minimization

problem (2.6) becomes,

$$\min_{\boldsymbol{\tau}_0 \in \mathscr{C}} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}_f - \boldsymbol{\tau}_0\|_{\boldsymbol{K}^{-1}}.$$

Solving this constrained minimization problem exactly will give the exact minimizer $\boldsymbol{\tau}_0^*$ and thus theoretically give the true error, which is an overkill in terms of finding a posteriori error estimator. In practice, we only need to solve it approximately since as long as $\boldsymbol{\tau} \in \mathscr{W}(\boldsymbol{f})$, $\psi(\boldsymbol{\tau})$ will provide an upper bound of the error, which can be used as an error estimator. Note that this approximation is (always) subject to a trade-off: the error estimator will approximate the true error very accurately if we solve the optimization problem at an unacceptable computation cost; or we can approximate and obtain a not-so-tight error estimator at an optimal computational cost.

2.2.2.1 Computing the Curl-free Component of the Error

In this subsection, we discuss how to compute $\boldsymbol{\tau}_f \in \mathscr{W}(\boldsymbol{f})$ with optimal computational complexity for a given graph. For any $\boldsymbol{\tau}_f \in \mathscr{W}(\boldsymbol{f})$, we have,

$$\boldsymbol{G}^\top \boldsymbol{\tau}_f = \boldsymbol{f}. \tag{2.7}$$

Since $\boldsymbol{G}^\top$ is the discrete divergence operator on the graph, the solution to the above equation is not unique and difficult to compute in general. However, we just need to find one $\boldsymbol{\tau}_f$. Here, based on a spanning tree $\mathcal{T}$ of the graph $\mathcal{G}$, we present an approach with optimal complexity, i.e., $\mathcal{O}(n)$ computational cost.

For a given spanning tree, $\mathcal{T} = (\mathcal{V}, \mathcal{E}_\mathcal{T}, \omega_\mathcal{T})$, we look for a $\boldsymbol{\tau}_f$ satisfying Equation (2.7) but that only has nonzero entries on the edges that belong to the spanning tree $\mathcal{T}$. In this case, we rewrite Equation (2.7) as:

$$\boldsymbol{f} = \boldsymbol{G}^\top \boldsymbol{\tau}_f = \begin{pmatrix} \boldsymbol{G}_\mathcal{T}^\top & \boldsymbol{G}_{\mathcal{G}\setminus\mathcal{T}}^\top \end{pmatrix} \begin{pmatrix} \boldsymbol{\tau}_{f,\mathcal{T}} \\ \boldsymbol{0} \end{pmatrix}, \tag{2.8}$$

where $\boldsymbol{G}_\mathcal{T}^\top$ is the discrete divergence operator that acts on edges in the tree $\mathcal{T}$, and

$\boldsymbol{G}_{\mathcal{G}\setminus\mathcal{T}}^{\top}$ is the discrete divergence operator on edges that are only in the graph $\mathcal{G}$ but not in the tree $\mathcal{T}$. From Equation (2.8), we have,

$$\boldsymbol{G}_{\mathcal{T}}^{\top}\boldsymbol{\tau}_{f,\mathcal{T}} = \boldsymbol{f}. \tag{2.9}$$

Therefore, once we solve Equation (2.9), we can assemble $\boldsymbol{\tau}_f$ by adding back the edges that are in the graph $\mathcal{G}$ but not in the tree $\mathcal{T}$. Note that Equation (2.9) is defined only on the spanning tree $\mathcal{T}$. We first solve,

$$\boldsymbol{L}_{\mathcal{T}}\boldsymbol{x} = \boldsymbol{f}, \tag{2.10}$$

$\boldsymbol{L}_{\mathcal{T}}$ being the graph Laplacian of the tree $\mathcal{T}$. With the fact that $\boldsymbol{L}_{\mathcal{T}} = \boldsymbol{G}_{\mathcal{T}}^{\top}\boldsymbol{K}_{\mathcal{T}}\boldsymbol{G}_{\mathcal{T}}$, where $\boldsymbol{K}_{\mathcal{T}}$ and $\boldsymbol{G}_{\mathcal{T}}$ are the discrete diagonal edge weight matrix and discrete gradient operator on tree $\mathcal{T}$, respectively, (2.10) is rewritten as follows,

$$\boldsymbol{G}_{\mathcal{T}}^{\top}\boldsymbol{K}_{\mathcal{T}}\boldsymbol{G}_{\mathcal{T}}\boldsymbol{x} = \boldsymbol{f}. \tag{2.11}$$

Comparing (2.9) and (2.11), we naturally have,

$$\boldsymbol{\tau}_{f,\mathcal{T}} \coloneqq \boldsymbol{K}_{\mathcal{T}}\boldsymbol{G}_{\mathcal{T}}\boldsymbol{x},$$

and thereafter assemble the full $\boldsymbol{\tau}_f = (\boldsymbol{\tau}_{f,\mathcal{T}}, \boldsymbol{0})^{\top}$. The procedure for computing $\boldsymbol{\tau}_f$ is summarized in Algorithm 3.

Algorithm 3 Computation of $\boldsymbol{\tau}_f$

1: **procedure** $[\boldsymbol{\tau}_f, \mathcal{T}]$ =COMPUTE$\boldsymbol{\tau}_f(\mathcal{G}, \boldsymbol{f})$
2: Build the spanning tree $\mathcal{T}$ from $\mathcal{G}$.
3: Solve $\boldsymbol{L}_{\mathcal{T}}\boldsymbol{x} = \boldsymbol{f}$, where $\boldsymbol{L}_{\mathcal{T}} = \boldsymbol{g}_{\mathcal{T}}^{\top}\boldsymbol{K}_{\mathcal{T}}\boldsymbol{G}_{\mathcal{T}}$.
4: Compute $\boldsymbol{\tau}_{f,\mathcal{T}} \leftarrow \boldsymbol{K}_{\mathcal{T}}\boldsymbol{G}_{\mathcal{T}}\boldsymbol{x}$.
5: Assemble $\boldsymbol{\tau}_f$ as $\boldsymbol{\tau}_f \leftarrow \begin{pmatrix} \boldsymbol{\tau}_{f,\mathcal{T}} \\ \boldsymbol{0} \end{pmatrix}$.
6: **end procedure**

Identifying a spanning tree via the classic Breadth First Search algorithm (BFS tree) takes $\mathcal{O}(m+n)$ computational complexity for general graphs [47]. If we consider

sparse graphs in which $m = \mathcal{O}(n)$ or $\mathcal{O}(n \log n)$, this step costs at most $\mathcal{O}(n \log n)$. The main computational cost of Algorithm 3 comes from Step 3, i.e., solving the linear system, Equation (2.10). As discussed in [113, 128], it takes linear time to solve (2.10). Additionally, the matrix-vector multiplication in Step 4 has $\mathcal{O}(n)$ complexity for sparse graphs. Therefore, the overall complexity of Algorithm 3 is at most $\mathcal{O}(n \log n)$.

2.2.2.2 Computing the the Div-free Component of the Error

For a given $\boldsymbol{\tau}_f$, we need to solve the following constrained minimization problem,

$$\boldsymbol{\tau}_0^* = \underset{\boldsymbol{\tau}_0 \in \mathscr{C}}{\operatorname{argmin}} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}_f - \boldsymbol{\tau}_0\|_{\boldsymbol{K}^{-1}}. \tag{2.12}$$

The difficulty here is that we need to satisfy the constraint exactly when we compute an approximate $\boldsymbol{\tau}_0^*$ to get the estimated value of the error. Our approach is to explicitly build the $(m - n + 1)$ basis, $\{\boldsymbol{c}^e\}$, of the cycle space $\mathscr{C}$ as discussed in Section 2.1 and transform the constrained minimization problem (2.12) into an unconstrained minimization problem. Denote by Ω the index set of the cycle basis. Then for any $\boldsymbol{\tau}_0 \in \mathscr{C}$, we write $\boldsymbol{\tau}_0$ as a linear combination of the cycle basis: $\boldsymbol{\tau}_0 = \sum_{e \in \Omega} \alpha_e \boldsymbol{c}^e$. Denote $\boldsymbol{\alpha} \in \mathbb{R}^{m-n+1}$, $(\boldsymbol{\alpha})_e = \alpha_e$. Thus, the minimization problem, (2.12), becomes,

$$\min_{\boldsymbol{\tau}_0 \in \mathscr{C}} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}_f - \boldsymbol{\tau}_0\|_{\boldsymbol{K}^{-1}} = \min_{\boldsymbol{\alpha}} \psi(\boldsymbol{\alpha}) = \min_{\alpha_e,\ e \in \Omega} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}_f - \sum_{e \in \Omega} \alpha_e \boldsymbol{c}^e\|_{\boldsymbol{K}^{-1}}. \tag{2.13}$$

This is an unconstrained least-squares problem and we can solve it with the usual approaches. Moreover, the approximate solution is guaranteed to belong to the cycle space. Solving (2.13) exactly eventually gives us the exact error, $\|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}}$. This step, however, has a computational complexity comparable to solving the original problem (2.2). Therefore, we solve it approximately via a few steps of a one-level overlapping Schwarz method (see [41, 124]). To describe this method, we first decompose the

cycle space, $\mathscr{C}$, into the following subspaces:

$$\mathscr{C} = \mathscr{C}_1 + \mathscr{C}_2 + \cdots + \mathscr{C}_J. \tag{2.14}$$

Note that $\mathscr{C}_i \cap \mathscr{C}_j$ is not necessarily empty. Then, each step of the Schwarz method corresponds to a loop over all subspaces and solving the minimization problem (2.13) in each of the subspace $\mathscr{C}_i$. That is, for $i = 1, 2, \cdots, J$, compute:

$$\min_{\Delta\boldsymbol{\tau}\in\mathscr{C}_{i+1}} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}_f - (\boldsymbol{\tau}_0^i + \Delta\boldsymbol{\tau})\|_{\boldsymbol{K}^{-1}}, \tag{2.15}$$

where $\boldsymbol{\tau}_0^i$ is the approximation to $\boldsymbol{\tau}_0^*$ after solving (2.15) in the first i subspaces, and $\boldsymbol{\tau}_0^* = \text{argmin}_{\boldsymbol{\tau}_0\in\mathscr{C}} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}_f - \boldsymbol{\tau}_0\|_{D^{-1}}$. The step of this relaxation method is completed after the approximation in $\mathscr{C}_J$ is computed. The solution obtained by iteratively solving equation (2.15) converges to the solution of the original minimization problem (2.12) since this can be interpreted as a subspace optimization method whose convergence property was discussed in [133].

To keep low computational cost in computing the error estimator, we only run $\mathcal{O}(1)$ iterations of Schwarz method to approximately solve (2.15) for $\boldsymbol{\tau}_0^*$. Later in section 2.3 we show that the error estimator computed with such an approximation is indeed accurate enough to capture the true error. The steps to compute $\boldsymbol{\tau}_0^* \in \mathscr{C}$ approximately are summarized in Algorithm 4, and we denote the approximation of $\boldsymbol{\tau}_0^*$ by $\boldsymbol{\tau}_0$.

Algorithm 4 Computing an Approximation to $\boldsymbol{\tau}_0^*$

procedure $\boldsymbol{\tau}_0$ =COMPUTE$\boldsymbol{\tau}_0(\mathcal{G}, \mathcal{T}, \boldsymbol{\tau}_f)$
  Build the cycle basis $\{\boldsymbol{c}^e\}$
  Given initial guess $\boldsymbol{\tau}_0^0 = \boldsymbol{0}$,
  for $i = 1, 2, \cdots,$ max_iter **do**
    for $k = 1, 2, \cdots, J$ **do** ▷ iterate over each subdomain
      $\Delta\boldsymbol{\tau}^* = \text{argmin}_{\boldsymbol{\eta}\in\mathscr{C}_k} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}_f + \boldsymbol{\tau}_0^{(i-1)J+k-1} + \boldsymbol{\eta}\|_{\boldsymbol{K}^{-1}}$.
      $\boldsymbol{\tau}_0^{(i-1)J+k} = \boldsymbol{\tau}_0^{(i-1)J+k-1} + \Delta\boldsymbol{\tau}$.
    end for
  end for
  return $\boldsymbol{\tau}_0^{J\cdot\text{max\_iter}}$.
end procedure

In Algorithm 4, the cost of one step of the Schwarz method depends on the number of subspaces J and the cost of solving (2.15) in each subspace. Here, we choose the following overlapping subspace decomposition: the i-the subspace is the span of the basis for the cycles incident with the vertex i. Thus, we have $J = n$, and

$$\mathscr{C}_i = \text{span}\{\boldsymbol{c}^j | \text{ cycle } j \text{ incidents with vertex } i\}, \quad i = 1, \ldots, J. \tag{2.17}$$

Since there are n vertices, we have $J = n$ subspaces. For sparse graphs using the special data structure proposed in [75], the solution of the minimization problem (2.15) on each subspace $\mathscr{C}_i$ is obtained with computational complexity at most $\mathcal{O}(\log n)$. This holds even in the rare cases when the dimension of $\mathscr{C}_i$ is large (e.g. $\sim n$). As a consequence, the overall computational cost of each iteration of Schwarz method is $\mathcal{O}(n \log n)$ for this choice of subspace decomposition (2.17), and this assures a low computational cost in computing the proposed estimator.

2.2.2.3 An Algorithm for A Posteriori Error Estimation with Helmholtz Decomposition

Now we are ready to present the overall Algorithm 5 to (approximately) solve the minimization problem (2.13) and compute a posteriori error estimation for solving the graph Laplacian (2.2).

Algorithm 5 Comptution of the error estimator $\min_{\boldsymbol{\tau}_0 \in \mathscr{C}} \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}_f - \boldsymbol{\tau}_0\|_{\boldsymbol{K}^{-1}}$

1: **procedure** ψ =ErrorEstimates($\mathcal{G}, \boldsymbol{v}, \boldsymbol{f}$)
2: $[\boldsymbol{\tau}_f, \mathcal{T}]$ = Compute $\boldsymbol{\tau}_f(\mathcal{G}, \boldsymbol{f})$.
3: $\boldsymbol{\tau}_0$ = Compute $\boldsymbol{\tau}_0(\mathcal{G}, \mathcal{T}, \boldsymbol{\tau}_f)$.
4: $\psi \leftarrow \|\boldsymbol{K}\boldsymbol{G}\boldsymbol{v} - \boldsymbol{\tau}_f - \boldsymbol{\tau}_0\|_{\boldsymbol{K}^{-1}}$. ▷ Compute the value of the estimator
5: **return** ψ.
6: **end procedure**

In Algorithm 5, Step 2 to compute $\boldsymbol{\tau}_f$ takes $\mathcal{O}(n \log n)$ for any sparse graph. Step 3 to compute $\boldsymbol{\tau}_0$ has complexity $\mathcal{O}(n \log n)$ for sparse graphs, since the minimization

problem (2.13) is solved approximately with $\mathcal{O}(1)$ iterations of Schwarz method. As a result, the overall computational complexity of Algorithm 5 is $\mathcal{O}(n \log n)$ for sparse graph $\mathcal{G}$.

To make the a posteriori error estimator more useful, especially for developing the adaptive AMG methods for solving graph Laplacians [83, 92, 135], we need to localize the a posteriori error estimator. Since,

$$\begin{aligned}\psi^2(\boldsymbol{\tau}) &= \|\boldsymbol{KGv} - \boldsymbol{\tau}\|^2_{\boldsymbol{K}^{-1}} = (\boldsymbol{KGv} - \boldsymbol{\tau})^T \boldsymbol{K}^{-1}(\boldsymbol{KGv} - \boldsymbol{\tau}) \\ &= \sum_{e \in \mathcal{E}} \frac{1}{\omega_e}\big((\boldsymbol{KGv} - \boldsymbol{\tau})e\big)^2,\end{aligned}$$

we localize the error estimator on each edge e as follows,

$$\psi_e(\boldsymbol{\tau}) = \boldsymbol{\omega}_e^{-\frac{1}{2}} |(DG\boldsymbol{v} - \boldsymbol{\tau})_e|. \tag{2.18}$$

We comment that the localized error estimators above are obtained for free in practice, since we have $(\boldsymbol{KGv} - \boldsymbol{\tau})_e$ available from the computation of the global error estimator $\psi(\boldsymbol{\tau})$ (see Step 4 in Algorithm 5).

This localized error estimator (2.18) then can be used to design adaptive AMG methods. For example, it can be utilized in generating coarser aggregations that approximate the fine aggregates (vertices) accurately [135] or generate approximations to the level sets of the error for the path cover adaptive AMG method [69].

2.3 Numerical Results

In this section, we present results of some numerical experiments demonstrating the efficiency of the a posteriori error estimator.

2.3.1 Tests on 2D Uniform Grids

We first test the performance of the algorithm on the unweighted graph Laplacian L of 2D uniform triangular grids, which corresponds to solving a Poisson equation on a 2D square domain with Neumann boundary condition. The uniform triangular grid

with grid size $h = 2^{-l}$, $l = 5, 6, 7, 8, 9$ is used, and we take $\boldsymbol{u} = \sin(\frac{\pi}{2}\boldsymbol{x})\sin(\frac{\pi}{2}\boldsymbol{y})$ as the exact solution. We set the approximate solution, $\boldsymbol{v} = \mathbf{0}$, and obtain the a posteriori error estimator $\psi(\boldsymbol{\tau})$ with Algorithm 5, in which the minimization problem (2.13) is solved approximately with several iterations of the overlapping Schwarz method. We use the face cycle bases that correspond to the small triangles in the grid (cycle length is 3). With this choice of cycle basis, each of the decomposed subspaces in (2.17) have dimension $\mathcal{O}(1)$ since there are at most six cycles incident from a given vertex i. The low dimension of the subspaces assures that solving (2.15) costs no more than $\mathcal{O}(1)$ computation and thus the computation cost of one iteration of Schwarz method remains $\mathcal{O}(n)$.

In Table 2.1, we report the exact error and the a posteriori error estimator $\psi(\boldsymbol{\tau})$ on graph Laplacian systems of different scales. Here, $e_{\mathit{ff}} \coloneqq \dfrac{\psi(\boldsymbol{\tau})}{\|\boldsymbol{u}-\boldsymbol{v}\|_{\boldsymbol{L}}}$ is also reported to show the efficiency of the error estimator. From 2.1, we observe that the CPU time for one iteration of the Schwarz method grows linearly as the size of graph Laplacian systems increases. The error estimator $\psi(\boldsymbol{\tau})$ gradually approaches the true error $\|\boldsymbol{u}-\boldsymbol{v}\|_L$ when we increase the steps of Schwarz iteration.

Table 2.1: Efficiency of the error estimator on graph Laplacian systems on uniform triangular grids of different sizes. The value of the estimator $\psi(\boldsymbol{\tau})$ is computed by solving (2.13) approximately with 1, 3, and 5 iterations of the overlapping Schwarz method. The CPU time (in seconds) is also shown in the table.

		1 iter			3 iters			5 iters		
$\lvert\mathcal{V}\rvert$	$\|\boldsymbol{u}-\boldsymbol{v}\|_{\boldsymbol{L}}$	$\psi(\boldsymbol{\tau})$	e_{ff}	time	$\psi(\boldsymbol{\tau})$	e_{ff}	time	$\psi(\boldsymbol{\tau})$	e_{ff}	time
1089	1.73	2.25	1.30	0.03	1.99	1.15	0.04	1.91	1.10	0.06
4225	1.73	2.67	1.55	0.05	2.28	1.32	0.11	2.16	1.25	0.16
16641	1.73	3.36	1.95	0.14	2.76	1.60	0.37	2.56	1.48	0.62
66049	1.72	4.43	2.57	0.53	3.51	2.03	1.40	3.20	1.86	2.31
263169	1.72	6.01	3.49	1.92	4.66	2.71	5.64	4.19	2.43	9.53

More importantly, we would like to know whether the localized error estimator (2.18) approximates the exact error on each edge accurately, since the localized

estimation is the key to an effective coarsening scheme in adaptive AMG. Take $\boldsymbol{L}$ as the weighted graph Laplacian of the uniform grid with grid size $h = 2^{-5}$, $\boldsymbol{u} = \sin(\frac{\pi}{2}\boldsymbol{x})\sin(\frac{\pi}{2}\boldsymbol{y})$, and $\boldsymbol{v}$ obtained by three iterations of the Gauss-Seidel method with a random initial guess. We compute the error estimator using three iterations of the Schwarz method to solve the minimization problem in Algorithm 5.

In Figure 2.2, we plot the difference between the exact error and the error estimator on each edge. On most of the edges the error estimator captures the true error well since the difference $\psi(\boldsymbol{\tau}) - \|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}}$ is no larger than 0.02.

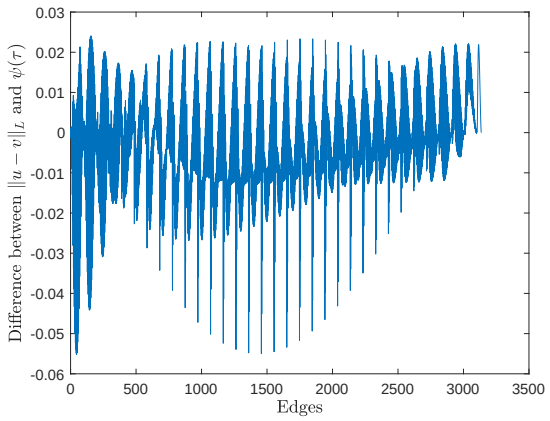


Figure 2.2: Difference between the true error $\|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}}$ and error estimator $\|\boldsymbol{KGv} - \boldsymbol{\tau}\|_{\boldsymbol{K}^{-1}}$ on each edge e.

2.3.2 Tests on "Real World" Graphs

In this section, we test the proposed error estimator on some "real-world" graphs from the SuiteSparse Matrix Collection [51]. We pre-process the undirected graphs by extracting the largest connected component of each graph and deleting self-loops. For each of these graphs, if the original edge weight is negative, we take its absolute value.

In Table 2.2, we summarize the basic information of the graphs and the performance of the error estimator. In our setting, $\boldsymbol{u}$ is the exact solution for a problem with arbitrarily chosen right-hand side $\boldsymbol{f}$. The approximate solution, $\boldsymbol{v}$, is obtained as a result of three iterations of the Gauss-Seidel method with this right-hand side. To compute the error estimator, we first use the breadth-first-search algorithm to

Table 2.2: Efficiency of error estimator on graph Laplacian systems arising from "real-world" applications. The value of the estimator $\psi(\boldsymbol{\tau})$ is computed by solving (2.13) approximately with 3 iterations of Schwarz method. The graph types tested are unweighted (u) and weighted (w).

ID	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	Problem Type	Type	$\lVert\boldsymbol{u}-\boldsymbol{v}\rVert_{\boldsymbol{L}}$	$\psi(\boldsymbol{\tau})$	e_{ff}
8	292	958	Least Squares Problem	u	1.74	1.75	1.00
1196	1879	5525	Circuit Simulation	w	2.71	2.71	1.00
22	5300	8271	Power Network	u	5.82	5.82	1.00
1614	2048	4034	Electromagnetic Problem	w	0.47	0.50	1.07
33	1423	16342	Structural Problem	w	14.5	19.7	1.36
791	8205	58681	Acoustic Problem	w	23.8	37.7	1.58
2777	1857	13762	Social Network	u	52.9	76.3	1.44
1533	2361	13828	Protein Network	u	4.61	4.70	1.01

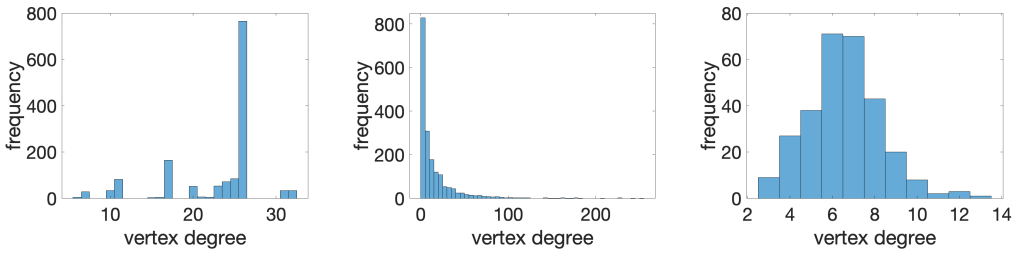


Figure 2.3: Representative degree distribution of networks studied in Table 2. *left*: Network ID 33. *middle*: Network ID: 2777 (power law distribution). *right:* Network ID 8 (normal distribution).

find a spanning tree and then construct the spanning-tree-induced fundamental cycle basis. Then, we apply three steps of Schwarz iterations to solve the minimization problem in Algorithm 5 to compute the overall error estimator. In Figure 2.3 we plot the degree distribution of selected networks. The differences in these degree distributions suggest that these networks have distinctive structural and dynamical properties. As we see from the results, for real-world graphs with different sizes, structures, and density, the error estimators approximate the exact errors well in all cases, which demonstrate the effectiveness of our proposed algorithm for computing the a posterior error estimator.

2.3.3 Tests on Building Aggregations for AMG

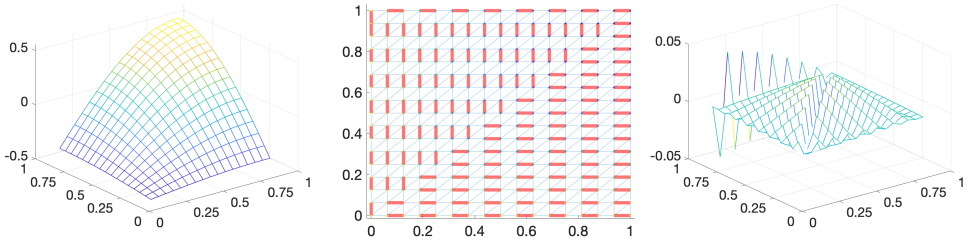
Designing an effective coarsening scheme that projects the smooth error onto coarse levels accurately is the key to the AMG methods. One example is the path cover adaptive algebraic multigrid (PC-αAMG) proposed in [69] for solving weighted graph Laplacian linear systems. Provided an accurate estimation of the current error, PC-αAMG first forms vertex-disjoint path cover where paths approximate the level sets of the smooth error, then aggregates vertices to form aggregations.

While an accurate error estimate makes PC-αAMG highly efficient, the step to compute an estimation of the exact error in the original PC-αAMG appears to be expensive in [69]. Our a posteriori error estimator offers an ideal replacement that gives an accurate approximation in an efficient manner.

Here, we present the resulting path-cover aggregation obtained by using the proposed a posteriori error estimator for solving the unweighted graph Laplacian on the 2D uniform grid. We solve the corresponding linear system approximately with several iterations of the relaxation scheme and curated the exact error of the current solution (plotted in Figure 4(a)). Note that in practice this exact error is not directly accessible, and we approximate it with the proposed a posteriori error estimator, plotted in Figure 4(d). In Figures 4(b) and 4(e), we compare the path-cover aggregation generated with the exact error and the a posteriori error estimator (computed approximately with 3 iterations of the Schwarz method). The aggregation

patterns are very similar.

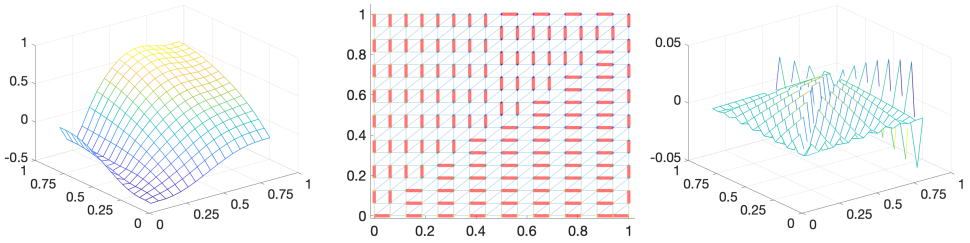
To check whether the smooth error is transferred and represented accurately on the coarse level, we restrict the smooth error to the coarse level and then prolongate it back. We plot the differences between the smooth error and error after restriction and prolongation in Figure 4(c) for the case where the coarse level is constructed based on the exact smooth error, and in Figure 4(f) for the case where the coarse level is constructed based on a posteriori error estimator, respectively. Although the shapes of differences in the two cases are different, the magnitudes of both cases are 0.045, which indicates that the aggregation built with the error estimator is effective in capturing the true smooth error.



(a) True error of the current solution.

(b) Path cover aggregation based on the known exact error.

(c) Difference between the smooth error and the error after restriction and prolongation with aggregation based on true error.



A posteriori error estimates of the current solution.

(e) Path cover aggregation based on a posteriori error estimator.

(f) Difference between the smooth error and the error after restriction and prolongation with aggregation based on error estimates.

Figure 4: Path cover aggregation generated with the exact error and a posteriori error estimates, and the difference between the smooth error and the error after restriction and prolongation. The aggregation based on the error estimator assembles the one based on the exact error, but the formal one is computable at a modest cost.

2.4 Alternative Formulation of Error Estimates

In [86, 87], the authors formulate a posteriori error estimators by relating to preconditioning in subspace correction methods. While these previous works are established within the field of Adaptive finite element methods (AFEMs), a natural extension exists for our applications on graph Laplacian systems. A few other AFEM error estimation techniques include but are not limited to, explicit residual estimators [129], implicit estimators based on local problems [9, 39, 99], hierarchical basis estimators [7, 8, 66], and equilibrated estimators [3, 19, 93].

We are interested in solving the following linear system with iterative methods,

$$\boldsymbol{L}\boldsymbol{u} = \boldsymbol{f},$$

where $\boldsymbol{L}$ is the weighted graph Laplacian of $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \omega)$. Given the current iterative solution $\boldsymbol{v}$, we want to approximate the error $\boldsymbol{e}^k = \boldsymbol{u} - \boldsymbol{v}$. In particular, we will derive a computable quantity $\psi(\boldsymbol{v})$ to provide a lower bound of the error, i.e.,

$$\|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}} \geq \psi(\boldsymbol{v}), \quad \forall \boldsymbol{v} \in \mathscr{V}.$$

For any $\boldsymbol{v} \in \mathscr{V}$, we can define an energy,

$$J(\boldsymbol{v}) = \frac{1}{2}\|\boldsymbol{v}\|_A^2 - (\boldsymbol{f}, \boldsymbol{v}).$$

It is straight forward to verify that $\boldsymbol{u} = \arg\min_{\boldsymbol{v}\in\mathscr{V}} J(\boldsymbol{v})$.

Lemma 2.4.1 *If $\boldsymbol{u}$ is the solution to $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{f}$, then for any $\boldsymbol{v} \in \mathscr{V}$,*

$$\|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}}^2 = 2(J(\boldsymbol{v}) - J(\boldsymbol{u})).$$

Proof:

$$2(J(\boldsymbol{v}) - J(\boldsymbol{u})) = \|\boldsymbol{v}\|_{\boldsymbol{L}}^2 - 2(\boldsymbol{f}, \boldsymbol{v}) - \|\boldsymbol{u}\|_{\boldsymbol{L}}^2 + 2(\boldsymbol{f}, \boldsymbol{u})$$

$$= \|\boldsymbol{v}\|_{\boldsymbol{L}}^2 - 2(\boldsymbol{L}\boldsymbol{u}, \boldsymbol{v}) - \|\boldsymbol{u}\|_{\boldsymbol{L}}^2 + 2(\boldsymbol{L}\boldsymbol{u}, \boldsymbol{u}) = \|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}}^2. \qquad \square$$

Lemma 2.4.2 *If $\boldsymbol{u}$ is the solution to $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{f}$, then for any fixed $\boldsymbol{v} \in \mathscr{V}$,*

$$\|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}}^2 \geq 2R_{\boldsymbol{v}}(\boldsymbol{w}) - \|\boldsymbol{w}\|_{\boldsymbol{L}}^2, \quad \forall \boldsymbol{w} \in \mathscr{V}, \tag{2.19}$$

where $R_{\boldsymbol{v}}(\boldsymbol{w}) = (\boldsymbol{f}, \boldsymbol{w}) - (\boldsymbol{L}\boldsymbol{v}, \boldsymbol{w})$.

Proof: Since $\boldsymbol{u} = \min_{\boldsymbol{v} \in \mathscr{V}} J(\boldsymbol{v})$, we have $J(\boldsymbol{u}) \leq J(\boldsymbol{v}+\boldsymbol{w})$ for any $\boldsymbol{w} \in \mathscr{V}$, and therefore,

$$\|\boldsymbol{u} - \boldsymbol{v}\|_{\boldsymbol{L}}^2 = 2(J(\boldsymbol{v}) - J(\boldsymbol{u})) \geq 2(J(\boldsymbol{v}) - J(\boldsymbol{v} + \boldsymbol{w})), \quad \forall \boldsymbol{w} \in \mathscr{V}.$$

Then,

$$\begin{aligned} 2(J(\boldsymbol{v}) - J(\boldsymbol{v} + \boldsymbol{w})) &= \|\boldsymbol{v}\|_{\boldsymbol{L}}^2 - 2(\boldsymbol{f}, \boldsymbol{v}) - \|\boldsymbol{v} + \boldsymbol{w}\|_{\boldsymbol{L}}^2 + 2(\boldsymbol{f}, \boldsymbol{v} + \boldsymbol{w}) \\ &= \|\boldsymbol{v}\|_{\boldsymbol{L}}^2 - 2(\boldsymbol{f}, \boldsymbol{v}) - \|\boldsymbol{v}\|_{\boldsymbol{L}}^2 - 2(\boldsymbol{L}\boldsymbol{v}, \boldsymbol{w}) - \|\boldsymbol{w}\|_{\boldsymbol{L}}^2 + 2(\boldsymbol{f}, \boldsymbol{v} + \boldsymbol{w}) \\ &= 2(\boldsymbol{f}, \boldsymbol{w}) - 2(\boldsymbol{L}\boldsymbol{v}, \boldsymbol{w}) - \|\boldsymbol{w}\|_{\boldsymbol{L}}^2 = 2R_{\boldsymbol{v}}(\boldsymbol{w}) - \|\boldsymbol{w}\|_{\boldsymbol{L}}^2. \qquad \square \end{aligned}$$

We set the right-hand side of equation (2.19) to be the lower bound error estimator, $\psi(\boldsymbol{v}) = 2R_{\boldsymbol{v}}(\boldsymbol{w}) - \|\boldsymbol{w}\|_{\boldsymbol{L}}^2$. To obtain a useful a posteriori error estimator, we need to maximize $\psi(\boldsymbol{v})$.

To show how well this error estimate approximates the exact error and the scalability of the computation, we repeat the numerical experiments with the same setting as in Section 2.3.1, with graph Laplacians of uniform triangular grids on a 2D square domain $[0,1]^2$. We take the exact solution, $\boldsymbol{u}$, to be $\boldsymbol{u} \sim \text{Unif}(0,1)$ and construct the right-hand side $\boldsymbol{f} = \boldsymbol{L}\boldsymbol{u}$. The approximate solution $\boldsymbol{v}$ is obtained with several iterations of the Gauss-Seidel method. We compute the error estimate $\psi(\boldsymbol{v})$ with the overlapping Schwarz method discussed in Section 2.2.2.2. Again, we choose the subspace to be the span of the fundamental basis for the cycles incident with each vertex, as defined in Equation (2.17). In Table 2.3, we report the exact error $\|\boldsymbol{u}-\boldsymbol{v}\|_{\boldsymbol{L}}$ and the a posteriori error estimator on $\psi(\boldsymbol{v})$ for graph Laplacian systems of different

scales. Again, $e_{\mathit{ff}} \coloneqq \dfrac{\psi(\boldsymbol{\tau})}{\|\boldsymbol{u}-\boldsymbol{v}\|_{\boldsymbol{L}}}$ is reported to show the efficiency of the error estimator. In Table 2.3, the error estimator approximates the exact error better when we increase the steps of the Schwarz iteration. In addition, the CPU time for one iteration of the Schwarz method grows linearly as the size of graph Laplacian systems increases.

Table 2.3: Efficiency of the error estimator on graph Laplacian systems on uniform triangle grids of different sizes. The value of the estimator $\psi(\boldsymbol{v})$ is computed approximately with 1,3, and 5 iterations of the overlapping Schwarz method. The CPU time (in seconds) is also shown in the table.

		1 iter			3 iters			5 iters		
$\lvert\mathcal{V}\rvert$	$\|\boldsymbol{u}-\boldsymbol{v}\|_A$	$\psi(\boldsymbol{\tau})$	e_{ff}	time	$\psi(\boldsymbol{\tau})$	e_{ff}	time	$\psi(\boldsymbol{\tau})$	e_{ff}	time
1089	5.70	5.29	0.93	0.02	5.66	0.99	0.07	5.69	1.00	0.12
4097	22.35	21.02	0.94	0.05	22.24	1	0.16	22.32	1.00	0.31
16641	2.28	1.45	0.64	0.29	2.76	0.91	0.52	2.19	0.96	1.54
66049	9.41	4.65	0.49	1.34	8.17	0.87	2.40	8.99	0.95	5.03

.

2.5 Conclusions

In this chapter, we proposed an a posteriori error estimator for solving linear systems of graph Laplacians. A novel approach is devised to reduce the computational cost of computing such an estimator in comparison to existing approaches. For sparse graphs, this novel estimator can be calculated in nearly-linear time. Our approach is based on the Helmholtz decomposition on the graphs. It includes solving a linear system on a spanning tree and solving (approximately) a minimization problem in the cycle space of the graph. We demonstrate how we can localize the error estimates on graph edges and use them to build high-quality AMG aggregation. However, the story is far from complete, and there is much room for exploring new techniques, especially given the wealth of work in adaptive AMG methods.

One future direction is to evaluate, theoretically or empirically, how the efficiency of our error estimators depends on the choice of spanning tree and the corresponding

cycle space basis [73] when we solve the graph Laplacians systems of graphs with different dynamical or structural properties.

Chapter 3

Application: Biological Protein-Protein Interaction Network

In the next two chapters, we elaborate on the two examples mentioned in Chapter 1 to illustrate the wide range of applications that give rise to the linear systems with graph Laplacians.

In this chapter, we present the application of graph Laplacian linear systems in studying protein-protein interaction networks. Unsmoothed Aggregation Algebraic Multigrid (UA-AMG) is used to efficiently compute the Diffusion State Distance (DSD) distance that captures complex structures in a manner robust to high-degree nodes in small-world biological networks. To further reduce the computational cost, we develop a dimension-reduction framework for the DSD, in which the data is projected into a small number of coordinates, such that Euclidean distances in the embedded space approximate DSD in the original space. The proposed algorithms are deployed for the analysis of real protein-protein interaction (PPI) networks with tasks like function prediction and link prediction.

3.1 Spectral Analysis of Diffusion State Distances

To study the spectral property of the DSD, we first generalize its definition in Chapter 1. Let $\ell^p(\boldsymbol{w})$ denote the space of p-integrable sequences with respect to the weight, $\boldsymbol{w}$. That is, for a fixed $\boldsymbol{w} = [w_1, \cdots, w_n]$, $\boldsymbol{x} = [x_1, \cdots, x_n] \in \ell^p(\boldsymbol{w})$ if $[\sum_{i=1}^n (x_i)^p w_i]^{\frac{1}{p}} < \infty$. The classic ℓ^p space is just $\ell^p = \ell^p(\mathbf{1})$. Then, the norm $\|x\|_{\ell^p(\boldsymbol{w})}$ is defined as the ℓ^p norm of x with respect to the weight $\boldsymbol{w}$,

$$\|\boldsymbol{x}\|_{\ell^p(\boldsymbol{w})} = \left[\sum_{i=1}^{n} (x_i)^p w_i\right]^{\frac{1}{p}}.$$

For example, $\|\boldsymbol{x}\|_p = \|\boldsymbol{x}\|_{\ell^p(\mathbf{1})}$, and $\|\boldsymbol{x}\|_{\ell^2(\boldsymbol{w})} = \sqrt{\sum_{i=1}^n (x_i)^2 w_i}$.

The q-step DSD between the vertex i and the vertex j defined in Equation (1.10) is generalized as,

$$\mathrm{DSD}^{q,p}_{\boldsymbol{w}}(i,j) := \|\boldsymbol{h}_i^q - \boldsymbol{h}_j^q\|_{\ell^p(\boldsymbol{w})},$$

and the DSD is generalized by letting $q \to \infty$, if this limit exists, i.e,

$$\mathrm{DSD}^{p}_{\boldsymbol{w}}(i,j) := \lim_{q\to\infty} \mathrm{DSD}^{q,p}_{\boldsymbol{w}}(i,j). \tag{3.1}$$

DSD can be defined in any ℓ^p space, $p \geq 1$. In Section 3.1 we focus on the case $p = 2$, in order to exploit the Hilbert space structure of ℓ^2. It is easy to see that $\boldsymbol{P}$ is diagonally conjugate to the symmetric matrix $\boldsymbol{D}^{\frac{1}{2}}\boldsymbol{P}\boldsymbol{D}^{-\frac{1}{2}} = \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}$. A computation [44] reveals that $\boldsymbol{P}$ may be written as a sum of outer products $\boldsymbol{P} = \sum_{\ell=1}^n \lambda_\ell \boldsymbol{\psi}_\ell^\top \boldsymbol{\varphi}_\ell$, where $(\boldsymbol{\psi}_\ell)_i = (\boldsymbol{\phi}_\ell)_i/\sqrt{\boldsymbol{\pi}_i}, (\boldsymbol{\varphi}_\ell)_i = (\boldsymbol{\phi}_\ell)_i\sqrt{\boldsymbol{\pi}_i}$, for $i = 1, \cdots, n$, and $\{(\lambda_\ell, \boldsymbol{\phi}_\ell)\}_{\ell=1}^n$ are the eigenvectors and eigenvalues of the symmetric matrix $\boldsymbol{D}^{\frac{1}{2}}\boldsymbol{P}\boldsymbol{D}^{-\frac{1}{2}}$. It follows that $\boldsymbol{P}^t = \sum_{\ell=1}^n \lambda_\ell^t \boldsymbol{\psi}_\ell^\top \boldsymbol{\varphi}_\ell$.

In addition, since we are working with a simple graph, $\mathcal{G}$, $\boldsymbol{P} = \boldsymbol{P}^\top$. In particular, $\{\boldsymbol{\phi}_\ell\}_{\ell=1}^n$ and $\{\boldsymbol{\varphi}_\ell\}_{\ell=1}^n$ are orthonormal bases for ℓ^2 and $\ell^2(1/\boldsymbol{\pi})$, respectively. With this decomposition, the DSD is re-formulated as follows.

Theorem 3.1.1 *[48] The diffusion state distance with weight $\boldsymbol{w} = 1/\boldsymbol{\pi}$ and $p = 2$ admits the decomposition $\mathrm{DSD}^2_{1/\pi}(i,j) = \sqrt{\sum_{\ell=1}^n (1-\lambda_\ell)^{-2}((\boldsymbol{\psi}_\ell)_i - (\boldsymbol{\psi}_\ell)_j)^2}$, where $\{(\lambda_\ell, \boldsymbol{\psi}_\ell)\}_{\ell=1}^n$ are the eigenvalues and right eigenvectors of $\boldsymbol{P}$.*

Proof: Decomposing DSD as a summation of $\boldsymbol{P}^t$, noting that $\boldsymbol{\psi}_1 = \mathbf{1}$, and summing over t as above yields,

$$\begin{aligned}
\mathrm{DSD}^2_{1/\boldsymbol{\pi}}(i,j) &= \left\|\boldsymbol{h}_i^\infty - \boldsymbol{h}_j^\infty\right\|_{\ell^2(1/\boldsymbol{\pi})} \\
&= \left\|\sum_{t=0}^\infty \boldsymbol{P}^t_{i\cdot} - \boldsymbol{P}^t_{j\cdot}\right\|_{\ell^2(1/\boldsymbol{\pi})} \\
&= \left\|\sum_{t=0}^\infty \sum_{\ell=1}^n \lambda_\ell^t(\psi_\ell)_i \boldsymbol{\varphi}_\ell - \lambda_\ell^t (\boldsymbol{\psi}_\ell)_j \boldsymbol{\varphi}_\ell\right\|_{\ell^2(1/\boldsymbol{\pi})}
\end{aligned}$$

$$= \left\| \sum_{t=0}^{\infty} \sum_{\ell=2}^{n} \lambda_\ell^t ((\boldsymbol{\psi}_\ell)_i - (\boldsymbol{\psi}_\ell)_j) \boldsymbol{\varphi}_\ell \right\|_{\ell^2(1/\boldsymbol{\pi})}$$
$$= \left\| \sum_{\ell=2}^{n} \sum_{t=0}^{\infty} \lambda_\ell^t ((\boldsymbol{\psi}_\ell)_i - (\boldsymbol{\psi}_\ell)_j) \boldsymbol{\varphi}_\ell \right\|_{\ell^2(1/\boldsymbol{\pi})}$$
$$= \left\| \sum_{\ell=2}^{n} \frac{1}{1-\lambda_\ell} ((\boldsymbol{\psi}_\ell)_i - (\boldsymbol{\psi}_\ell)_j) \boldsymbol{\varphi}_\ell \right\|_{\ell^2(1/\boldsymbol{\pi})}$$

Recalling that $\{\boldsymbol{\varphi}_\ell\}_{\ell=1}^n$ is an orthonormal basis for $\ell^2(1/\boldsymbol{\pi})$, the result follows from Parseval's theorem. □

3.1.1 Diffusion State Distance and Dimension Reduction

Theorem 3.1.1 implies that the DSD is in fact the Euclidean distance in a new coordinate basis, given by the transformation,

$$x_i \mapsto ((1-\lambda_2)^{-1}(\boldsymbol{\psi}_2)_i, (1-\lambda_3)^{-1}(\boldsymbol{\psi}_3)_i, \cdots, (1-\lambda_n)^{-1}(\boldsymbol{\psi}_n)_i).$$

This representation naturally lends itself to dimension reduction in the following sense. Suppose that for some M, $1/(1-\lambda_M) \gg 1/(1-\lambda_{M+1})$. Then the truncated representation,

$$x_i \mapsto \left(\frac{1}{1-\lambda_2}(\boldsymbol{\psi}_2)_i, \frac{1}{1-\lambda_3}(\boldsymbol{\psi}_3)_i, \cdots, \frac{1}{1-\lambda_M}(\boldsymbol{\psi}_M)_i \right) \tag{3.2}$$

, is faithful in the sense that the discarded coordinates are comparatively insignificant. Moreover, depending on the number of data points, the spectral decomposition is unreliably noisy after some $M \ll n$ [63], in which case the truncation of the coordinate change has the added benefit of denoising. This approach to computing the DSD is an alternative to randomized approaches [90].

To understand the condition $1/(1-\lambda_M) \gg 1/(1-\lambda_{M+1})$, let $\lambda_\ell = 1-\mu_\ell$ for $0 = \mu_1 \leq \mu_2 \leq \cdots \leq \mu_n$. Then the condition becomes $\frac{1}{\mu_M} \gg \frac{1}{\mu_{M+1}}$, which is a measure of a gap in the near-reducibility of certain subsets of the underlying Markov transition matrix [97]. Indeed, suppose that $\lambda_M = 1-e^{-q_1}, \lambda_{M+1} = 1-e^{-q_2}$ for $q_1 \geq q_2 \geq 0$. If $q_1 = q_2$, then the corresponding eigenvectors $\boldsymbol{\psi}_M, \boldsymbol{\psi}_{M+1}$ count equally in the DSD,

since the weights $1/(1-\lambda_\ell), \ell = M, M+1$ are equal. On the other hand, if $q_1 \gg q_2$, then there is a large gap between μ_M and μ_{M+1}: $\mu_M/\mu_{M+1} = e^{-q_1+q_2} \ll 1$; in this case, the DSD may be truncated after the M^{th} eigenvector while preserving much of the information present in the full DSD. We remark that using only a small number of eigenvectors to compute data-dependent distances is common in the context of spectral clustering [105, 118].

3.1.2 Relationship Between Diffusion State Distances and Inverse Laplacians

As shown in Chapter 1 Theorem 1.2.1, the DSD is closely related to the inverse of a regularized Laplacian $(\boldsymbol{I} - \boldsymbol{P}^\top + \boldsymbol{W}^\top)^{-1}$ [15, 38]. It is enlightening to consider the Neumann series expansion, noting that $\lambda_1 = 1, \boldsymbol{\psi}_1 = \mathbf{1}, \boldsymbol{\varphi}_1 = \boldsymbol{\pi}^\top$:

$$
\begin{aligned}
(\boldsymbol{I} - \boldsymbol{P}^\top + \boldsymbol{W}^\top)^{-1} &= \left(\boldsymbol{I} - \left(\boldsymbol{P}^\top - \boldsymbol{\pi}\mathbf{1}^\top\right)\right)^{-1} \\
&= \sum_{t=0}^{\infty} (\boldsymbol{P}^\top - \boldsymbol{\pi}\mathbf{1}^\top)^t \\
&= \sum_{t=0}^{\infty} ((\boldsymbol{P}^\top)^t - \boldsymbol{\pi}\mathbf{1}^\top) \\
&= \sum_{t=0}^{\infty} \left(\left(\sum_{\ell=1}^{n} \lambda_\ell^t \boldsymbol{\varphi}_\ell^\top \boldsymbol{\psi}_\ell\right) - \boldsymbol{\pi}\mathbf{1}^\top\right) \\
&= \sum_{t=0}^{\infty} \left(\sum_{\ell=2}^{n} \lambda_\ell^t \boldsymbol{\varphi}_\ell^\top \boldsymbol{\psi}_\ell\right) \\
&= \sum_{\ell=2}^{n} \left(\sum_{t=0}^{\infty} \lambda_\ell^t\right) \boldsymbol{\varphi}_\ell^\top \boldsymbol{\psi}_\ell \\
&= \sum_{\ell=2}^{n} \frac{1}{1-\lambda_\ell} \boldsymbol{\varphi}_\ell^\top \boldsymbol{\psi}_\ell.
\end{aligned}
$$

Hence, $DSD_w^p(i,j) = \| \sum_{\ell=2}^{n} \frac{1}{1-\lambda_\ell} (\boldsymbol{\psi}_\ell(i) - \boldsymbol{\psi}_\ell(j)) \boldsymbol{\varphi}_\ell(\cdot) \|_{\ell^p(w)}$. Noting that $\{\boldsymbol{\varphi}_\ell\}_{\ell=1}^n$ is an orthonormal basis for $\ell^2(1/\boldsymbol{\pi})$, we see that this expansion is of particular use when $w = 1/\boldsymbol{\pi}, p = 2$. We remark that the original formulation of DSD used the weight $w = \mathbf{1}$, i.e. a constant weight. If the degrees of the underlying graph generating P are nearly constant, then the difference between this original formulation and the

DSD with weight $1/\boldsymbol{\pi}$ is small. The more disparate the degree structure, the more the $1/\boldsymbol{\pi}$ factor in the proposed DSD will correct for this degree imbalance.

3.1.3 Computational Considerations

For a dense graph with n nodes, naive direct computation of all pairwise DSD is $O(n^3)$, which is unacceptably inefficient for large datasets. Recall that the right eigenvectors $\{\boldsymbol{\psi}_\ell\}_{\ell=1}^n$ of $\boldsymbol{P}$ are, after scaling by $\sqrt{\boldsymbol{\pi}}$, eigenvectors of the symmetric normalized Laplacian $\boldsymbol{N} = \boldsymbol{I} - \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}$ with corresponding eigenvalues $\mu_\ell = 1 - \lambda_\ell$. We thus compute the DSD using $\boldsymbol{N}$ according to Algorithm 6.

In Algorithm 6, we first compute $\boldsymbol{Y}$, which are the coordinates suggested in Section 3.1.1. Moreover, we further apply the dimension reduction technique discussed in Section 3.1.1, which implies that we only compute the first M eigenpairs and then form a truncated version of $\boldsymbol{Y}$. This is summarized in Algorithm 7.

Algorithm 6 Compute DSD

1: Compute the eigenpairs $\{(\mu_\ell, \boldsymbol{\phi}_\ell)\}_{\ell=1}^n$ of the symmetric normalized graph Laplacian $\boldsymbol{N}$.
2: Compute $\boldsymbol{Y} = \boldsymbol{\Sigma}\boldsymbol{\Phi}\boldsymbol{D}^{-\frac{1}{2}}$, where $\boldsymbol{\Sigma} = \mathrm{diag}(\mu_2^{-1}, \mu_3^{-1}, \cdots, \mu_n^{-1})$ and $\boldsymbol{\Phi} = (\boldsymbol{\phi}_2, \boldsymbol{\phi}_3, \cdots, \boldsymbol{\phi}_n)$.
3: Compute $\mathrm{DSD}^2_{1/\boldsymbol{\pi}}(i,j) = \|\boldsymbol{Y}(:,i) - \boldsymbol{Y}(:,j)\|_2$.

Algorithm 7 Compute approximate DSD

1: Compute the first M eigenpairs $\{(\mu_\ell, \boldsymbol{\phi}_\ell)\}_{\ell=1}^M$, of $\boldsymbol{N}$.
2: Compute $\widetilde{\boldsymbol{Y}} = \boldsymbol{\Sigma}_M\boldsymbol{\Phi}_M\boldsymbol{D}^{-\frac{1}{2}}$, where $\boldsymbol{\Sigma}_M = \mathrm{diag}(\mu_2^{-1}, \mu_3^{-1}, \cdots, \mu_M^{-1})$, $\boldsymbol{\Phi}_M = (\boldsymbol{\phi}_2, \boldsymbol{\phi}_3, \cdots, \boldsymbol{\phi}_M)$.
3: Compute $\widetilde{\mathrm{DSD}}^2_{1/\boldsymbol{\pi}}(i,j) = \|\widetilde{\boldsymbol{Y}}(:,i) - \widetilde{\boldsymbol{Y}}(:,j)\|_2$.

The main computational cost in both algorithms is the first step, i.e., computing the eigenpairs of $\boldsymbol{N}$. To do this, we use a preconditioned Krylov-based eigensolver, such as the implementation in ARPACK [84]. The performance of such a Krylov-based eigensolver depends on the choices of the preconditioner. Since we are working with the normalized graph Laplacian, the algebraic multigrid method (AMG) is used as a preconditioner [134]. More precisely, we use the aggregation-based AMG

method, which is a suitable choice for solving graph Laplacian problems as shown in [24, 69, 90, 92] due to its (nearly) optimal computational complexity in practice. Indeed, in the case of dense graphs, Algorithm 7 has complexity $O(n^2)$, while in the case of sparse graphs, complexity quasilinear in n can be achieved through fast matrix-vector multipliers. In both cases, the complexity of Algorithm 7 is the same as for computing the spectral embedding in classical spectral clustering [105].

3.2 Computational and Numerical Experiments on Biological Networks

In this section, we present how to use DSD for the analysis of real biological networks. In order to demonstrate the denoising and the computational speed-ups, throughout this section we consider DSD in ℓ^2, particularly $\mathrm{DSD}^2_{1/\boldsymbol{\pi}}$.

3.2.1 Information about the Data Sets

The DREAM Disease Module Identification Challenge [43] released a heterogeneous collection of six different human protein-protein association networks with different biological criteria for placing edges between different protein pairs, in order to benchmark different methods for unsupervised network clustering. Here we consider the first two of these networks, DREAM1 and DREAM2.

DREAM1 is derived from STRING [122], which integrates known and predicted protein-protein associations from various sources, including those aggregated from databases that collect experimental evidence for protein-protein interactions, those derived from co-expression, and others either inferred by literature annotation or transferred from interactions of sequence-similar proteins in other species. DREAM1 includes all the different interaction types in STRING v.10.0 excluding the interactions derived from text mining. Edge weights correspond to the STRING association score (after removing evidence from text mining).

DREAM2 intends to represent a more classical protein-protein interaction network, where the presence of an edge between proteins indicates that there is evidence

	Vertices	Edges	Average Degree
DREAM1	17,388	2,232,398	128.4
DREAM2	12,325	397,254	32.2

Table 3.1: Number of vertices, number of edges, and average degrees for the largest connected components in the DREAM1 and DREAM2 networks.

that proteins physically bind in the cell. It is derived from the InWeb database [85], where edges come from either humans or are supported by the presence of a corresponding interaction in multiple different model organism databases (InWeb requires each such interaction in four separate model organism databases before it includes it in its network as also a human interaction edge with confidence). These networks are dense and highly connected (see Table 3.1), though they can have some isolated nodes. To ensure the assumption of irreducibility for convergence, we restrict to the largest connected component of the network if it is not originally connected. The edge weights reflect the confidence of the interaction (See Table 3.1).

To illustrate the value of DSD for the biological networks, as well as the virtue of the DSD approximation via eigenvector truncation, we consider two classical problems in computational biology: *link prediction* and *function prediction*.

3.2.2 Link Prediction

Despite a large number of edges, it is assumed that many remaining true edges are missing from the PPI networks, because some of the underlying interactions have not yet been experimentally observed. The goal of *link prediction* is to estimate these missing links, based on the already observed ones. A solution to the link prediction problem is an ordered ranking of all possible pairs of vertices that are not directly connected by an edge in the network. A classifier predicts the edges ranked above a threshold to be "true edges" and below the threshold to be "non-edges" where varying the threshold allows a tradeoff between true and false positive edge predictions.

Both the DREAM1 and the DREAM2 networks are very large and have dense, highly connected cores. Because of these densely connected cores, the overwhelming majority of missing links in the graphs are between nodes in the core region. As

the core is highly connected, a simple heuristic measure of ranking the node pairs based on the number of common neighbors will produce an excellent ranking when applied to the entire network, since we expect every node pair in the core to have some neighbors in common. However, we are often most interested in predicting new interactions outside the common core. In order to avoid simply discovering the common core, and also to work with smaller and more computationally tractable examples, we consider the link prediction problem on sparser and smaller samples of the DREAM1 and DREAM2 networks. More precisely, we randomly sample 400 nodes from the network, and consider the "full" network of their induced subgraph. We then compute a "partial" network, consisting of the same subgraph of nodes with 10% of the edges removed uniformly at random. While removing the edges, we ensure graph connectedness using Algorithm 8.

Algorithm 8 Generate random connected subgraphs

1: Input a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \omega)$ as a list of edges, with $m = |\mathcal{E}|$.
2: Randomly permute the edge indices $\{1, \ldots, m\}$.
3: Create a spanning tree from this ordered edge list, prioritizing the edges with smaller assigned numbers.
4: Protecting the edges of the resulting tree, remove $0.10m$ of the edges from the remaining graph uniformly at random.

We note that, starting from the smaller 400-node networks, the method of removing some of the known links to be predicted is the same as the method employed by [54]. In a real biological application, the 400 nodes would not be chosen at random, but would rather be a set of genes used in a Y2H experiment [30], or known to be associated with some disease. The retention of the spanning tree automatically means we retain a connected network and are not trying to predict new edges to isolated nodes, which is impossible for any PPI embedding method in the absence of additional biological information beyond the input graph. This models the idea that some "true" links are missing from the known network because experimental evidence is incomplete.

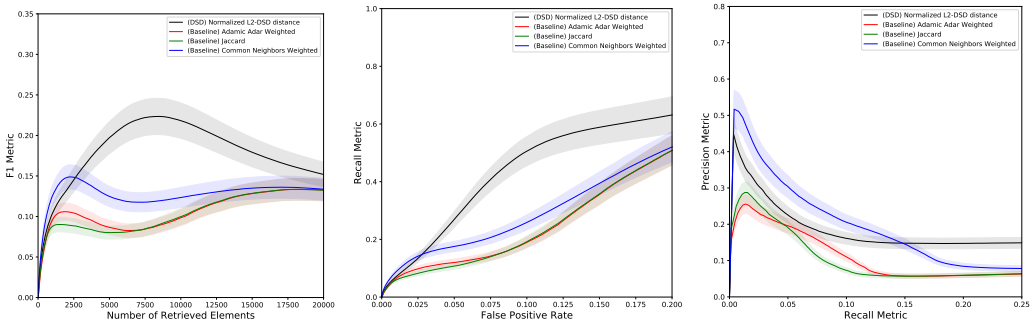
Link prediction ranking methods are then compared against each other by starting with the partial network, and assuming the missing edges from the full network

Score	Definition
Weighted Common Neighbor	$\sum_{x_k \in \mathcal{N}(x_i) \cap \mathcal{N}(x_j)} (\boldsymbol{A}_{ik} + \boldsymbol{A}_{jk})$
Jaccard's Coefficient	$\lvert\mathcal{N}(x_i) \cap \mathcal{N}(x_j)\rvert / \lvert\mathcal{N}(x_i) \cup \mathcal{N}(x_j)\rvert$
Weighted Adamic-Adar Index	$\sum_{x_k \in \mathcal{N}(x_i) \cap \mathcal{N}(x_j)} 1/\log(1 + \sum_{x_\ell \in \mathcal{N}(y)} \boldsymbol{A}_{\ell k})$

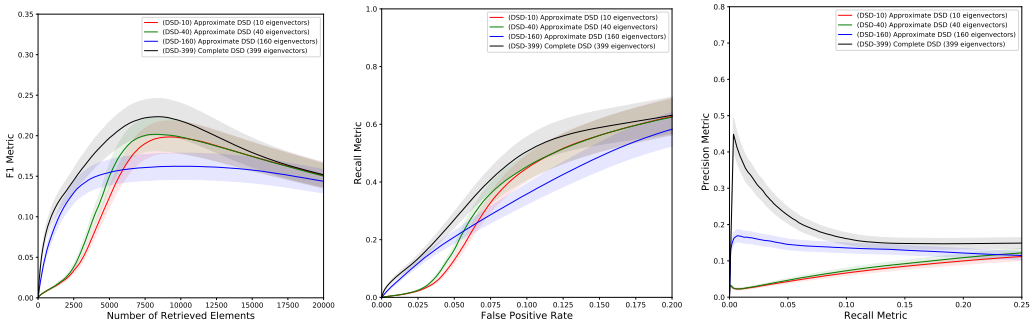
Table 3.2: Definitions of different heuristic method scores for two nodes $x_i, x_j \in X$ in an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \omega)$ with neighbor sets $\mathcal{N}(x_i)$ and $\mathcal{N}(x_j)$ respectively.

are the true positive links, and all unobserved links from the full network are negative. In this sense, we understand the partial network as a training set, and the goal is to predict the full network. DSD-based link prediction is done by ranking every node pair in the graph in ascending order based on their DSD distances, with the interpretation that the lower the DSD distance between two nodes in the graph, the more likely the nodes are to have a link between them. The heuristic comparison methods considered—weighted common neighbor, Jaccard's coefficient, and the weighted Adamic-Adar index [89]—are local scoring methods, where the link scores between two distinct nodes in the graph are computed based on their neighborhood affinity. Since a larger score implies that the two nodes are closely associated in the network, a link between them becomes more likely. By ranking all the edges in the ascending order of the score and providing a threshold, we get a score-based method that predicts the missing link in the graph. Table 3.2 shows how the heuristic scores are computed for two arbitrary nodes $x_i, x_j \in X$ in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \omega)$.

Figures 3.1 and 3.2 are obtained by averaging the results generated from 100 iterations of sampling uniformly at random subgraphs of 400 nodes from the DREAM1 and DREAM2 networks and generating training and test sets, as described in Algorithm 8. DSD-based ranking is compared against the heuristic methods and diffusion distances. By varying the threshold and mapping the true positive versus the false positive rate of the various methods, we compute the F1 score. Similarly, we construct the partial receiver operating characteristic (ROC) [59] and precision-recall curves from the top 20000 ranked edges for the reduced graphs. The ROC plots show



(a) F1, heuristics (b) ROC, heuristics (c) Precision-recall, heuristics

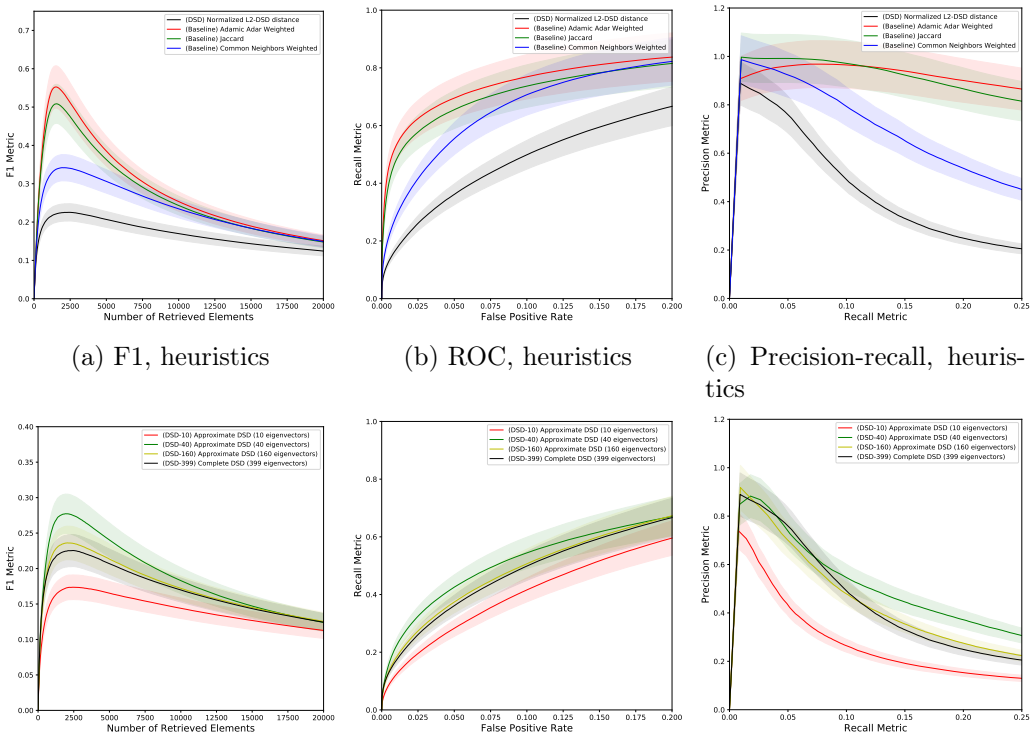


(d) F1, Reduced DSD (e) ROC, Reduced DSD (f) Precision-recall, Reduced DSD

Figure 3.1: *(a), (b), (c):* Comparison of DSD to the different heuristic ranking methods on 100 randomly sampled graphs from DREAM1, each having 400 nodes. On all of the average F1, ROC and precision-recall curves, we see DSD outperforms the heuristic methods. *(d), (e), (f):* Comparison of DSD to the approximate DSD methods on 100 randomly sampled graphs from DREAM1, each having 400 nodes. Limiting the number of eigenvectors used significantly decreased the run time while not substantially degrading the empirical performance. **Note:** The shaded regions demarcate plus and minus one standard deviation of the score.

that the top edges predicted by DSD-based ranking are more accurate, compared to both the heuristic methods and the diffusion distance based methods for DREAM1. In DREAM2, the heuristic methods slightly outperform the DSD method.

Figures 3.1 and 3.2 (g-i) show the results from using approximate DSD computations in which the spectral decomposition is truncated to denoise and reduce computational complexity; see (3.2). We find that approximate DSD is nearly comparable to exact DSD in performance in DREAM1; in DREAM2, it improves link prediction performance. For both DREAM1 and DREAM2, approximate DSD significantly decreases run time.



(a) F1, heuristics (b) ROC, heuristics (c) Precision-recall, heuristics

(d) F1, Reduced DSD (e) ROC, Reduced DSD (f) Precision-recall, Reduced DSD

Figure 3.2: *(a), (b), (c):* Comparison of DSD to the different heuristic ranking methods on 100 randomly sampled graphs from DREAM2, each having 400 nodes. On all of the average F1, ROC and precision-recall curves, we see DSD is outperformed by the heuristic methods. *(d), (e), (f):* Comparison of DSD to the approximate DSD methods on 100 randomly sampled graphs from DREAM2, each having 400 nodes. Limiting the number of eigenvectors improves results while also lowering computational complexity. **Note:** The shaded regions demarcate plus and minus one standard deviation of the score.

3.2.3 Function Prediction

Protein-protein association networks can assist in making predictions for functional roles of unknown proteins, since proteins with similar functions should cluster (under the appropriate metric) in the network. DSD was introduced in the context of function prediction [38] and it was shown that predicting protein function based on a majority vote of the most popular function among the k-nearest neighbors (kNN) of a node in DSD distance performed well in a yeast protein-protein interaction network. Here, we evaluate DSD for function prediction in the DREAM1 and DREAM2 human networks. Function labels are taken from the popular Gene Ontology (GO) database [46] (downloaded from FuncAssociate3.0 [14] on 02/12/19), where we restrict the GO labels we consider in our cross-validation experiments to those that are neither too general nor too specific. In particular, we consider all GO function labels from the Biological Process hierarchy and Molecular Function hierarchy that annotate between 100 and 500 proteins in the networks.

On the largest connected components of the biological networks, we use a five-fold cross-validation to demonstrate the accuracy rate of function prediction of Gene Ontology (GO) labels based on the DSD metric computed as in Algorithm 6. For each node in the testing set, we generate its kNN based on the DSD metric, then use majority voting weighted by $\frac{1}{\mathrm{DSD}^2_{1/\boldsymbol{\pi}}(x_i, x_j)}$ to predict its function; let $\eta(x)$ denote the predicted function of protein x. The functional label for x is marked correct if $\eta(x)$ is among the functional labels assigned to the node in the test set. We report the percent of nodes assigned a correct functional label as the accuracy score. This is summarized in Algorithm 9.

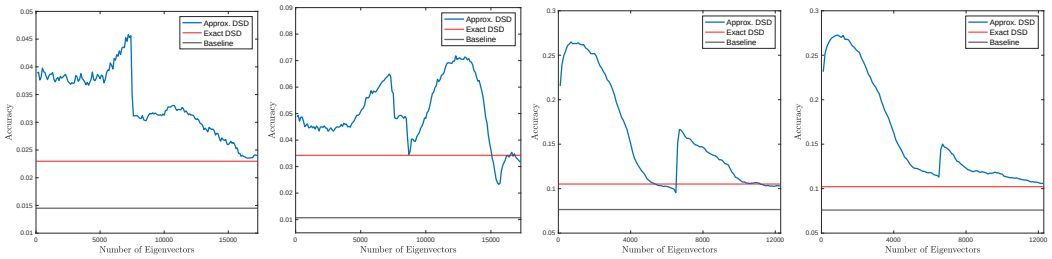
We investigate the accuracy and efficiency of function prediction on the same networks with approximate DSD computed by Algorithm 7 and compare the numerical result to the one computed with exact DSD, using $k = 10$ nearest neighbors. We compute the approximate DSD using the MATLAB build-in eigensovler `eigs`, which

Algorithm 9 Function prediction with majority voting

1: **for** every node x in the testing set **do**
2: Compute its DSD kNN ($k = 10$) neighbors $\{v_i\}_{i=1}^{k}$.
3: Each neighbor that is also in the training set votes for its GO function labels, weighted by $1/\mathcal{D}^2_{1/\pi}(x, v_i)$.
4: Use the most voted label as the predicted function label, denoted by $\eta(x)$.
5: Mark the node correct if $\eta(x)$ is one of the labels of x.
6: **end for**
7: Compute the accuracy score as the percentage of nodes that get a correct functional label.

is based on ARPACK [84].

In Figure 3.3, the accuracy of function prediction of DREAM1 and DREAM2 networks based on approximated DSD with different numbers of eigenvectors is compared to the accuracy result based on exact DSD (the red horizontal line). Function prediction with the exact or approximated DSD metric demonstrates a substantial increase in accuracy compared to the best-studied baseline method, majority vote [38], where all direct neighbors in the original graph vote for their functional labels with equal weight for each label (the grey horizontal line). Compared to the exact DSD metric, the accuracy of the approximated DSD metric initially increases as we include more eigenvectors, and it almost always outperforms exact DSD for both DREAM1 and DREAM2 networks, and also for labels in both the molecular function and biological process GO hierarchies. After that, the prediction accuracy decreases in all cases when we include the high-frequency eigenvectors, which are possibly corrupted by noise. We note that the approximate DSD methods we employ are not immune to numerical error, which prevents the approximate DSD from perfectly converging to the exact DSD when all the eigenvalues are used (see the numerical instability at the far right of each of the graphs of Figure 3.3). These numerical errors in the eigendecomposition depend on the machine epsilon, the condition number of $\boldsymbol{P}$, the magnitude of its eigenvalues, as well as the stopping criterion for the eigensolver, and are most acute for the highest frequency eigenvectors. Thus when we truncate the number of eigenvectors used to the lowest frequency eigenvectors, the numerical instability is much less of a factor, and the same picture would likely be observed if



(a) DREAM1, Biological Process Labels

(b) DREAM1, Molecular Function Labels

(c) DREAM2, Biological Process Labels

(d) DREAM2, Molecular Function Labels

Figure 3.3: Percentage of the accuracy of function prediction for DREAM1 and DREAM2 networks using the GO Biological Process hierarchy and the Molecular Function hierarchy labels by exact DSD and approximated DSD. The improved function prediction results illustrate the efficiency in denoising the DSD metric by using only the top eigenvectors. We note that both exact and approximate DSD-based function prediction strongly outperform the baseline method.

	Exact DSD	Approximate DSD
DREAM1	17084.17	3090.918
DREAM2	2759.05	87.049

Table 3.3: CPU time in seconds used to compute exact DSD and approximate DSD for each DREAM network. The approximate DSD for DREAM1 is computed with the first 6000 eigenvectors and the one for DREAM2 uses the first 1000 eigenvectors. The computation of the approximate DSD is much faster, since only a small number of eigenvectors of $\boldsymbol{P}$ (or equivalently, of $\boldsymbol{N}$) need to be computed.

we were computing exact DSD projected to the lower-dimensional eigenspace.

From Figure 3.3 the performance of function prediction peaks when we restrict to the first 6000 eigenvectors to compute the approximate DSD for DREAM1 network with labels in the Biological Process hierarchy, and there is a peak also at the first 6000 eigenvectors but a slightly higher second peak at around 12000 eigenvectors for the Molecular Function hierarchy. For DREAM2, restricting to the first 1000 eigenvectors for both gives the best performance. With only these eigenvectors we are able to compute an accurate low-dimensional embedding of the DSD metric, reflected by the function prediction performance in Figure 3.3. We compare the CPU time for both approaches in Table 3.3 and it is observed that computing the exact DSD metric is much more computationally expensive.

3.3 Conclusions and Future Directions

In this chapter, we exploit the spectral decompositions of the underlying random walk of diffusion state distances to reduce computational complexity and denoise. Experiments on synthetic and real data illustrate that the DSD is typically more effective for link prediction and function prediction tasks, compared to classical graph-based metrics and also to heuristic methods used for biological network analysis. Moreover, the spectral formulation of DSD is shown not only to improve efficiency, but accuracy in link and function prediction by denoising.

From the computational biology perspective, random-walk based measures of proximity have long been understood to illuminate relationships in the networks that are studied [49]. DSD measures, in particular, have recently been increasingly adopted in a variety of settings, from de-noising networks for the study of cancer driver genes [68], to hierarchical clustering in the latest version of the STRING network [123], as well as featuring prominently in the DREAM Disease Module challenge itself as both a stand-alone method and as the basis of a novel consensus method to integrate diverse clusterings [43]. In addition to inference based on single sources of network data, as described in this chapter, how best to take advantage of multiplex networks remains an active area of research [43, 137]. It would also be interesting to explore further the approximate DSD measures described in this chapter, and understand how to automatically choose the right number of dimensions, based on simple parameters of the network structure such as network density. Moreover, DSD-based link prediction performs somewhat poorly on the DREAM2 network, compared to its performance in link prediction on DREAM1, and its function prediction performance. This is hypothesized to be due to the especially dense core of DREAM2. Developing methods that interpolate between the heuristic methods (which perform well on the dense core) and DSD-based methods (which perform well off the dense core) is the topic of ongoing research [54].

Chapter 4

Application: Graph Learning

In this chapter, we present the applications of graph Laplacian systems in a few machine learning problems. In particular, as discussed in Section 1.2.2, we focus on the analysis and algorithms of Laplace learning, Poisson learning, and p-Laplace learning, and look into the learning cases with very low label rate. In Section 4.1 and 4.2 we provide several sets of experiments to illustrate the efficiency of these graph learning algorithms under different scenarios, followed by the analysis of their learning behaviors. We conclude the chapter and discuss the future work from both the theoretical and the application aspects in Section 4.3.

4.1 Label Classification on Biology Networks

The set of six biological networks we experimented with is the ones used in [42], from the STRING database v9.1 [62] without text-mining. These networks share a set of nodes but each has its own set of edges and edge weights. The nodes correspond to genes that appear in the union of all the networks. The edges correspond to a different type of relationship between pairs of genes for each network. As pointed out in [42], these relations range from experimental evidence of interaction or association between two genes, to co-expression, among other things. Finally, the weights indicate the confidence in the edges being correct. These six heterogeneous networks contain over $6,400$ genes with the number of edges varying from $1,361$ to $314,013$ for yeast, and $18,362$ genes with the number of edges varying from $3,717$ to $1,544,348$ for human.

We use the GO functional labels as described in Section 3.2.3. The GO functional labels are grouped into three distinct functional hierarchies: Biological Process(BP), Molecular Function(MF), and Cellular Component(CC), where we mimic the setup in [42] to filter the GO terms to only retain those of intermediate specificity, labeling

more than 10 and less than 301 genes among the nodes. This label set can be further divided into levels of varying specificity, each containing labels that annotate 11-30, 31-100, and 101-300 genes, respectively [115], filtering on the Jaccard index to remove too similar GO labels. We consider 9 different functional annotation experiments, parameterized by one of the 3 hierarchies (BP, MF, and CC) times one of the three levels of specificity of GO terms (11-30, 31-100, and 101-300).

In [42], the Mashup algorithm for function prediction by integrating information across multiplex biological networks consists of the following three core steps:

1. **Diffusion.** On each network, a diffusion process is run which creates a Random-Walk-with-Restart (RWR) matrix representation of the network.

2. **Embedding.** A shared embedding is created using the matrix representations generated in the diffusion step. This is achieved via a singular value decomposition or dictionary learning techniques. Ultimately this gives a low-dimensional vector representation of every node in the dataset.

3. **Learning.** Once every node in the dataset is represented by a vector, existing function prediction methods can be applied using the embedding and the available annotations.

We note there are many ways of generating node embeddings on graphs, such as Diffusion State Distance [37, 38], Node2vec [65], and spectral methods [88]. In [42] a Random-Walk-with-Restart (RWR) based approach [80] is adopted. For each of the network $\mathcal{G}^i, i = 1, \cdots, 6$, with n nodes, for each vertex $u \in \mathcal{G}^i$, Mashup iteratively computes the t-step RWR distribution $\boldsymbol{s}_u^t \in \mathbb{R}^n$ as follows. With restart rate $0 \leq \alpha \leq 1$, let $\boldsymbol{s}_u^0 = \boldsymbol{e}_u$,

$$\boldsymbol{s}_u^{t+1} = (1-\alpha)\boldsymbol{P}^i \boldsymbol{s}_u^t + \alpha \boldsymbol{e}_u, \tag{4.1}$$

where $\boldsymbol{P}$ is the transition probability matrix defined in Equation (1.8), and $\boldsymbol{e}_u \in \mathbb{R}^n$ is the vector with entry 1 at the u-th index and 0 elsewhere. Following [37, 38, 42], the node embedding of u is the *diffusion state* $\boldsymbol{s}_u^\infty$, the stationary distribution at the fixed point of the iteration (4.1). By stacking the diffusion states $\boldsymbol{s}_u^\infty$ as columns, we

obtain the diffusion state matrix $\boldsymbol{S}^i$, which is the RWR representation matrix used in Mashup.

In Step 2, a low-dimensional embedding is learned to canonically represent the topological patterns in the six individual networks. We can view the learned embedding as a new graph, where each node is assigned a new coordinate represented by its embedding vector. The weight of the edge connecting two nodes is assigned to be the reciprocal of ℓ_2 distance between the embedding vectors of the two nodes. Therefore, the closer the two nodes are, the heavier the weight of the edge that connects them. Then, we replace Step 3 by testing our graph learning algorithms on this new embedding graph. In addition, to illustrate the merits of Poisson learning and p-Laplace learning in the learning tasks with few labels, we test with cases where only $1, 2, 4, 8$, and 16 labeled nodes per class are used in the training, besides the cases where we use all the available labels in the training set. In Table 4.1 and Table 4.2, we present the accuracy of function label prediction carried by different learning algorithms: majority voting with 10 nearest neighbors weighted by edge weights, Laplace learning, Poisson learning, and p-Laplace learning with $p = 4$. The predicted label of a node is marked correct if it matches the functional labels of that node in the test set, and the accuracy score measures the percent of nodes that get assigned a correct functional label.

Yeast BP 31-100						
# of labeled nodes	1	2	4	8	16	all
Majority Voting	6.32%	9.68%	14.90%	22.73%	30.28%	51.37%
Laplace	4.73%	9.82%	18.04%	29.31%	36.48%	55.90%
Poisson	17.57%	22.31%	29.61%	37.65%	43.40%	54.78%
p-Laplace ($p = 4$)	13.08%	19.00%	26.38%	34.11%	40.99%	57.37%

Table 4.1: Function prediction performance on Yeast PPI network with different graph learning algorithms in 5-fold cross-validation. GO labels from the BP hierarchy with a term range of $31 - 100$ are used.

In Table 4.1 and Table 4.2, we observe that in the tests with sparse labels, Poisson learning outperforms the other learning algorithms, followed by p-Laplace learning in the second place. The gap between the two winning algorithms and the

Human MF 31-100						
# of labeled nodes	1	2	4	8	16	all
Majority Voting	4.68%	6.69%	10.25%	18.07%	24.32%	40.22%
Laplace	3.04%	3.23%	7.45%	16.58%	24.95%	39.62%
Poisson	12.66%	16.18%	21.20%	26.77%	33.95%	46.61%
p-Laplace ($p = 4$)	10.27%	13.35%	16.23%	21.18%	29.47%	47.12%

Table 4.2: Function prediction performance on Human PPI network with different graph learning algorithms in 5-fold cross-validation. GO labels from the MF hierarchy with a term range of $31 - 100$ are used.

vanilla weighted majority voting or Laplace learning is larger when the labels are extremely sparse (1 or 2 labels per class). When all labels in the training set are used, p-Laplace learning surpasses Poisson learning. However, we must point out that when $p \geq 3$, the nonlinear p-Laplace learning is always computationally more expensive than Poisson learning or standard Laplace learning. More discussion on the computation of p-Laplace learning will be covered in Section 4.2.

In Section 1.2.1 and Chapter 3, we discussed how DSD can help us extract important data features in the biological graphs. Here, we test how DSD impacts these downstream graph learning algorithms as well. In particular, since [34] proposed a random-walk interpretation of Poisson and Laplace learning, we check whether these graph learning algorithms can (partially) replace the DSD computation by extracting latent network structure based on a variation of graph diffusion on these networks. In Table 4.3 and Table 4.4, we examine how the different learning algorithms perform on both the original biological networks and the diffused networks. That is, we first experiment with biological graphs with their original edge weights that reflect the confidence in the edges being correct, and then test with the edge weights assigned as the reciprocal of the diffusion state distance between pairs of nodes. We benchmark the performance of the graph learning algorithms with the weighted majority voting (WMV) by 10 nearest neighbors. Again, besides using all the available labels in the training set, we curate learning cases where only $1, 2, 4, 8,$, and 16 labeled nodes per class are included, so that we can check the learning behavior on sparsely labeled graphs. This time we test on a single biological network to avoid any confounding

factors that might arise in a multi-network learning task. The following tests are run on the co-expression network from [42], for both the Yeast and the Human PPI network.

Yeast BP 11-30						
# of labeled nodes	1	2	4	8	16	all
Original + WMV	2.51%	4.50%	7.19%	9.82%	12.71%	13.56%
Original + Laplace	7.30%	10.14%	12.56%	16.21%	18.82%	19.38%
Original + Poisson	9.50%	12.23%	14.77%	17.24%	15.34%	2.75%
DSD + WMV	4.07%	6.30%	9.55%	13.29%	17.10%	17.60%
DSD + Laplace	10.15%	12.44%	14.85%	19.18%	21.15%	23.09%
DSD + Poisson	10.64%	12.88%	14.93%	18.80%	20.00%	20.17%

Table 4.3: Function prediction performance of the graph learning algorithms on both the original Yeast gene co-expression network and the diffused network with 5-fold cross-validation. GO labels from the BP hierarchy with a term range of $11-30$ are used.

Human MF 11-30						
# of labeled nodes	1	2	4	8	16	all
Original + WMV	1.12%	1.91%	3.28%	5.03%	6.76%	6.85%
Original + Laplace	2.29%	4.44%	7.87%	9.37%	11.52%	11.90%
Original + Poisson	5.51%	7.13%	9.32%	11.77%	10.04%	3.61%
DSD + WMV	1.82%	3.19%	5.76%	8.20%	12.04%	13.12%
DSD + Laplace	4.94%	7.18%	9.79%	12.02%	14.01%	15.09%
DSD + Poisson	6.52%	7.89%	9.50%	11.27%	13.88%	13.85%

Table 4.4: Function prediction performance of the graph learning algorithms on both the original Human gene co-expression network and the diffused network with 5-fold cross-validation. GO labels from the MF hierarchy with a term range of $11-30$ are used.

From Table 4.3 and Table 4.4, it is observed that in all cases the additional step to compute DSD helps improve function prediction performance, which further supports our belief in DSD to capture the complex structure in biological networks. Poisson learning still outperforms Laplace learning and weighted majority voting in the cases with very few labels, with or without the DSD computation. However, we note the strange catastrophic performance of Poisson learning when applied directly to the original network with all the training labels used, compared to the relatively robust performance of Laplace Learning.

4.2 Algorithms for p-Laplace Learning

As aforementioned in Section 1.2.2.2, we choose to compute p-Laplace learning with Newton's method. Finding a good initial guess point is very important for the efficiency of Newton's methods, since if we reach the quadratic convergence region, Newton's iteration should converge in only a few steps [71]. Here we discuss two approaches in an effort to find good initial guesses; the first approach involves a homotopy method in p, and the second one is a nested iteration approach.

4.2.1 Newton's Method with Homotopy in p

Following the suggestion in [112], we can significantly speed up Newton's method to solve Equation (1.27) with a continuation in p. To be more specific, to compute the solution Equation (1.27) with large p, we will compute the solution for an increasing sequence of values of p starting at $p = 2$. In this way, each time we use the solution from the previous value of p as the initial guess for the next minimization. This initial guess falls in the quadratic convergence region if each increment in the sequence of values of p is small enough. As a result, at each increment in p, the computation only requires a modest number of Newton steps.

We test p-Laplace learning using Newton's method with homotopy on the following graph learning problem. We use the weighted graph of 2D uniform triangluar grids on $[0,1] \times [0,1]$, where the edge weights are randomly generated. The grid graph is of size 30×30 with 900 nodes. We set $\boldsymbol{u} = \sin(\boldsymbol{x})\cos(\boldsymbol{y})$ and compute the source term $\boldsymbol{f}(\boldsymbol{x}) = -\Delta_p^G \boldsymbol{u}(\boldsymbol{x})$. Recall that in Equation (1.31), a graph Laplacian needs to be inverted in each Newton iteration. Instead of using the conjugate gradient method(CG), preconditioned with an Incomplete Cholesky factorization as suggested in [112], we use an AMG solver to invert the graph Laplacian. We first solve the minimization problem with $p = 2$, and use the solution of $p = 2$ as the initial solution for the $p = 3$ problem. We take increasingly large steps in p and apply homotopy until $p = 50$ is reached. For each value of p, we solve the problem with Newton's method up to the tolerance $\epsilon = 10^{-7}$, where ϵ is the residual norm. In Figure 4.1 we

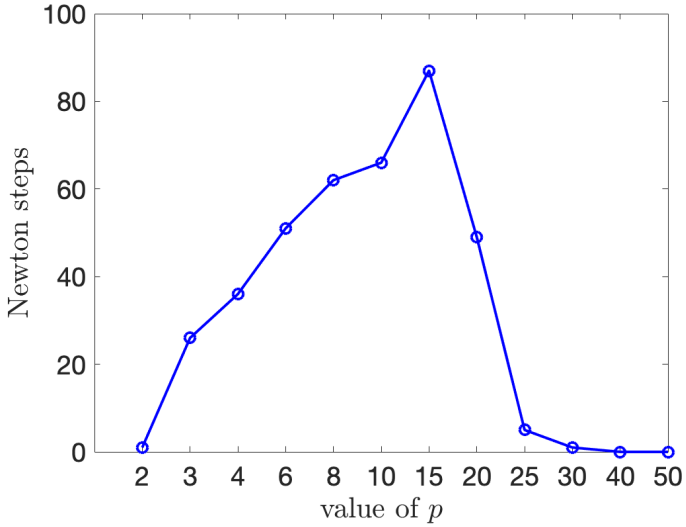


Figure 4.1: Number of Newton steps used when we use homotopy in p.

show the number of Newton steps used before the tolerance is reached. We observe that with homotopy, the new initial guess remains near the convergence regime and Newton's method converges every time. Note that when p increases from 2 to 20, the numbers of Newton steps increase, since each sequential problem with higher p is more nonlinear and more difficult to solve. Afterward, it takes significantly fewer steps to converge at each increment. We numerically verified that, as p becomes very large, the solutions to Equation (1.31) become close to each other, while the solution tends to change more drastically when p values are small. There is much room to explore this behavior theoretically in future works.

4.2.2 Nested Iteration for p-Laplace Learning

In this section, we discuss ways to solve systems of equations with graph Laplacians efficiently by using a multilevel algorithm called *nested iteration*, so that most of the work is done on coarse grids where computation is much cheaper. The general idea of nested iteration goes as follows. First, we solve the p-Laplace learning on a coarse version of the original graph/grids, where we linearize the problem, and solve it with Newton's method based on a random initial guess. When the Newton steps converge and we solve the problem "well enough" on the coarse grid, we interpolate

this solution to a finer graph as its initial guess. We then linearize and solve the problem on the finer graph with Newton's method again, and repeat the process until we solve the original learning problem. We describe a two-level scheme here and note that this can be generalized to a multi-level scheme easily.

Set the original graph $\mathcal{G}$ to be the fine level graph $\mathcal{G}_f = (\mathcal{V}_f, \mathcal{E}_f, \boldsymbol{\omega}_f)$, and the denote the coarse level graph as $\mathcal{G}_c = (\mathcal{V}_c, \mathcal{E}_c, \boldsymbol{\omega}_c)$. The approximation to the solution $\boldsymbol{u}$ from the fine and coarse grids are denoted as $\boldsymbol{u}_f$ and $\boldsymbol{u}_c$, respectively. And the minimization problems on fine and coarse grids are denoted as $J_p^f(\cdot)$ and $J_p^c(\cdot)$, respectively. To define the coarse-level graph, we need to explicitly find $\mathcal{V}_c, \mathcal{E}_c, \boldsymbol{\omega}_c$. To obtain the vertex set $\mathcal{V}_c$, we unsmoothed aggregation on the unlabeled (interior) fine-level nodes and keep the labeled (boundary) nodes as they are. Denote the number of aggregates formed during the unsmoothed aggregation by n_c. Then, on the coarse level there are n_c unlabeled nodes and m labeled nodes, $|\mathcal{V}_c| = n_c + m$. we use $\boldsymbol{R}$ to represent the restriction matrix that maps from the fine grid to the coarse grid, $\boldsymbol{R} \in \mathbb{R}^{|\mathcal{V}_c| \times |\mathcal{V}_f|}$. With an appropriate ordering of the coarse nodes, the restriction operator is defined blockwise as,

$$\boldsymbol{R} = \begin{pmatrix} \boldsymbol{R}_{11} & 0 \\ 0 & \boldsymbol{R}_{22} \end{pmatrix} \begin{matrix} \longrightarrow \text{unlabeled nodes} \\ \longrightarrow \text{labeled nodes} \end{matrix}$$

, where $\boldsymbol{R}_{11} \in \mathbb{R}^{n_c \times n}$ is the unsmoothed aggregation restriction operator of the labeled nodes, and $\boldsymbol{R}_{22} \in \mathbb{R}^{m \times m}$ is the identity matrix. We find the set of edges, $\mathcal{E}_c$, and set of edge weights, $\boldsymbol{\omega}_c$, on the coarse graph by computing the Laplacian of the coarse graph. Following the definition (1.7), let $\boldsymbol{L}_f \in \mathbb{R}^{|\mathcal{V}_f| \times |\mathcal{V}_f|}$ denote the graph Laplacian of $\mathcal{G}_f$. Then, the Laplacian of the coarse graph, $\boldsymbol{L}_c \in \mathbb{R}^{|\mathcal{V}_c| \times |\mathcal{V}_c|}$, is defined by,

$$\boldsymbol{L}_c = \boldsymbol{R}\boldsymbol{L}\boldsymbol{R}^\top.$$

After that, we can assemble $\mathcal{E}_c$ and $\boldsymbol{\omega}_c$ from $\boldsymbol{L}_c$ according to definition (1.7).

The key to designing an efficient nested iteration scheme is to define the coarse-level problem that gives rise to a good approximation of the original and the fine-level solutions. On the fine level, the optimization problem is,

$$\begin{aligned} J_p^f(\boldsymbol{u}_f) &= (\boldsymbol{u}_f, \Delta_p^G(\boldsymbol{u}_f)) + (\boldsymbol{u}_f, \boldsymbol{f}_f) \\ &= (\boldsymbol{u}_f, \boldsymbol{G}^\top \boldsymbol{K}(\|\boldsymbol{G}\boldsymbol{u}_f\|^{p-2}\boldsymbol{G}\boldsymbol{u}_f)) + (\boldsymbol{u}_f, \boldsymbol{f}_f). \end{aligned} \tag{4.3}$$

We present two ways to define the coarse-level problem.

Algorithm 1: To define the coarse level problem, we first define coarse graph operators $\boldsymbol{G}_c, \boldsymbol{K}_c, \boldsymbol{\mathcal{L}}_c, \boldsymbol{B}_c$ following the definitions in (1.5) and (1.29). Since the unsmoothed aggregation coarsening tends to preserve the total connection between aggregations as well as the graph structure of the fine graph, we are motivated to define the coarse-level problem as a straightforward analogy of the fine-level problem,

$$\begin{aligned} \min_{\boldsymbol{u}_c} J_p^{c_1}(\boldsymbol{u}_c) &= (\boldsymbol{u}_c, \Delta_p^{G_c}(\boldsymbol{u}_c)) + (\boldsymbol{u}_c, \boldsymbol{f}_c) \\ &= (\boldsymbol{u}_c, \boldsymbol{G}_c^\top \boldsymbol{K}_c(\|\boldsymbol{G}_c\boldsymbol{u}_c\|^{p-2}\boldsymbol{G}_c\boldsymbol{u}_c)) + (\boldsymbol{u}_c, \boldsymbol{f}_c). \end{aligned} \tag{4.5}$$

Note that this coarse-level problem can be solved with Newton's method, just like how the minimization is solved on the fine grid. The Newton iteration is,

$$\boldsymbol{u}_c^{k+1} = \frac{p-2}{p-1}\boldsymbol{u}_c^k + \frac{1}{p-1}\boldsymbol{\mathcal{L}}_c(\boldsymbol{u}_c^k)^{-1}\left[\boldsymbol{B}_c(\boldsymbol{u}_c^k)\boldsymbol{g} - \boldsymbol{f}\right]$$

Computational-wise, the inversion of the Laplacian matrix $\boldsymbol{\mathcal{L}}_c(\boldsymbol{u}_c^k)$, the bottleneck of the computational cost, is executed for each Newton iteration on the coarse level first. We then prolongate this solution $\boldsymbol{u}_c$ to the fine level, with the hope that $\boldsymbol{R}^\top \boldsymbol{u}_c$ serves as a reasonably good initial guess for the fine-level problem. In this way, we can perform fewer steps of Newton's iteration on the fine level, where each iteration is more computationally expensive.

However, the efficiency of this nested iteration algorithm is affected by the fact that the coarse-level problem $J_p^{c_1}(\cdot)$ may not approximate the fine-level problem $J_p^f(\cdot)$ very well. Both minimization problems are defined as the sum of graph gradients

weighted by edge weights, but the edge sets and edge weights are changed after the coarsening. Therefore, we consider the second approach that computes the coarse-level problem while factoring in the original set of edge and edge weights.

Algorithm 2: Alternatively, we can define a different coarse problem as,

$$\begin{aligned}\min_{\boldsymbol{u}_c} J_p^{c_2}(\boldsymbol{R}^\top \boldsymbol{u}_c) &= (\boldsymbol{R}^\top \boldsymbol{u}_c, \Delta_p^G(\boldsymbol{R}^\top \boldsymbol{u}_c) + (\boldsymbol{R}^\top \boldsymbol{u}_c, \boldsymbol{f}_f) \\ &= (\boldsymbol{R}^\top \boldsymbol{u}_c, \boldsymbol{G}^\top \boldsymbol{K}(\|\boldsymbol{G}\boldsymbol{R}^\top \boldsymbol{u}_c\|^{p-2}\boldsymbol{G}\boldsymbol{R}^\top \boldsymbol{u}_c)) + (\boldsymbol{R}^\top \boldsymbol{u}_c, \boldsymbol{f}_f).\end{aligned} \tag{4.7}$$

Note that this is essentially the fine-level minimization except the argument is defined in the range of $\boldsymbol{R}^\top$. Encoding the known labels and simplifying in the same way as in equation (1.26), we end up working with $\min_{\boldsymbol{u}_c} J_p^{c_2}(\boldsymbol{R}_{11}^\top \boldsymbol{u}_c)$. Compute the gradient $\nabla J_p^{c_2}(\boldsymbol{R}_{11}^\top \boldsymbol{u}_c) \in \mathbb{R}^{n_c}$ and Hessian $\nabla^2 J_p^{c_2}(\boldsymbol{R}_{11}^\top \boldsymbol{u}_c) \in \mathbb{R}^{n_c \times n_c}$ with respect to $\boldsymbol{u}_c$ to get,

$$\begin{aligned}\nabla J_p^{c_2}(\boldsymbol{R}_{11}^\top \boldsymbol{u}_c) &= \boldsymbol{R}_{11}\boldsymbol{L}(\boldsymbol{R}_{11}^\top \boldsymbol{u}_c)\boldsymbol{R}_{11}^\top \boldsymbol{u}_c - \boldsymbol{R}_{11}\boldsymbol{B}(\boldsymbol{R}_{11}^\top \boldsymbol{u}_c)\boldsymbol{g}_f + \boldsymbol{R}_{11}\boldsymbol{f}_f, \\ \nabla^2 J_p^{c_2}(\boldsymbol{R}_{11}^\top \boldsymbol{u}_c) &= (p-1)\boldsymbol{R}_{11}\boldsymbol{L}(\boldsymbol{R}_{11}^\top \boldsymbol{u}_c)\boldsymbol{R}_{11}^\top.\end{aligned}$$

And the Newton iteration to update $\boldsymbol{u}_c$ becomes:

$$\begin{aligned}\boldsymbol{u}_c^{k+1} &= \boldsymbol{u}_c^k - \left[\nabla^2 J_p^{c_2}(\boldsymbol{R}_{11}^\top \boldsymbol{u}_c^k)\right]^{-1} \nabla J_p^{c_2}(\boldsymbol{R}_{11}^\top \boldsymbol{u}_c^k) \\ &= \frac{p-2}{p-1}\boldsymbol{u}_c^k + \frac{1}{p-1}\left[\boldsymbol{R}_{11}\boldsymbol{L}(\boldsymbol{R}_{11}^\top \boldsymbol{u}_c^k)\boldsymbol{R}_{11}^\top\right]^{-1} \boldsymbol{R}_{11}\left[\boldsymbol{B}(\boldsymbol{R}_{11}^\top \boldsymbol{u}_c^k)\boldsymbol{g}_f - \boldsymbol{f}_f\right].\end{aligned} \tag{4.9}$$

We consider the following numerical example to demonstrate the potential of Algorithm 2. We use the weighted graph of 2D uniform triangular grids on $[0,1] \times [0,1]$, and the edge weights are randomly generated. The uniform triangular grid of size $h = 2^l, l = 4,5,6,7$ is used. We set $\boldsymbol{u} = \sin(\frac{\pi \boldsymbol{x}}{2})\cos(\frac{\pi \boldsymbol{y}}{w})$ and compute the source term $\boldsymbol{f}(\boldsymbol{x}) = -\Delta_p^G \boldsymbol{u}(\boldsymbol{x})$. The boundary nodes are set to be the ones with coordinates $x = 0, x = 1, y = 0$, or $y = 1$. We report the behavior of the two-level nested iteration in table 4.5. $p = 10$ for p-Laplace learning. We report the number of Newton steps used on the coarse and fine grid until the residual norm reaches the tolerance of $\epsilon = 10^{-7}$, as well as the value of the functional $J_p^{c_2}(\cdot)$ and $J_p^f(\cdot)$ when the tolerance

is reached. We include in the table the heuristic true value of the functional, $J^f(\boldsymbol{u})$, for comparison. The maximum Newton iterations run on each level is set to be 100.

Table 4.5: Efficiency of the nested iteration in solving p-Laplace learning on graphs of uniform triangular grids with different sizes. The number of Newton iterations and the converged functional values $J_p^{c_2}(\cdot)$ and $J_p^f(\cdot)$ are reported in the table.

			Nested Iteration Coarse Level		Nested Iteration Fine Level		Direct Compute Fine Level	
$\lvert\mathcal{V}_f\rvert$	$\lvert\mathcal{V}_c\rvert$	$J^f(\boldsymbol{u})$	# Steps	$J_p^{c_2}(\cdot)$	# Steps	$J_p^f(\cdot)$	# Steps	$J_p^f(\cdot)$
256	149	-129.82	77	-128.06	44	-129.82	75	-129.82
1024	532	-504.31	60	-502.04	50	504.31	96	-504.31
4096	2015	-2073.50	66	-2071.08	48	-2073.50	100	-2073.48
16384	7771	-8427.03	92	-8425.10	55	-8427.03	100	-8426.97

From table 4.5, we see that the number of Newton steps required to reach residual error tolerance on the fine level with nested iteration is much smaller compared to the number of Newton steps used when we start the computation directly on the fine level with a random initial guess. We believe the work done on the coarse grid gives a good approximation, and therefore, a good initial guess, of the solution on the fine grid. When the tolerance is reached, $J_p^{c_2}(\cdot)$ is always greater than $J_p^f(\cdot)$ due to the discretization error.

In addition, by defining the coarse-level problem in this way, we essentially compute the same functional $J_p(\cdot)$ (with respect to the original set of edges and edge weights) but with an argument lying within the range of $\boldsymbol{R}^\top$. We are able to track the changes in the functional during the solving phase on both the coarse and fine levels. We note that the functional $J_p^f(\boldsymbol{R}^\top \boldsymbol{u}_c^k)$ decreases as we update of coarse solution $\boldsymbol{u}_c^k$. This guarantees consistency in solving this minimization problem, whereas Algorithm 1 fails to provide such monotonicity in minimizing the functional.

4.3 Conclusion and Future Work

In this chapter, we study the applications of several graph learning algorithms that are mathematically equivalent to solving one or a sequence of graph Laplacian systems. We work with their applications on the biological networks, and explored the learning cases where only a very limited number of nodes are labeled. We also introduce the technique of Newton's method with homotopy and nested iteration to accelerate the computation of p-Laplace learning. Note that a common goal to use either homotopy or nested iteration is to find a good initial guess for Newton's method. In addition, nested iteration reduces the computational cost by completing most of the work on coarse grids where computation is relatively cheap. In future works, we plan to first generalize the two-level algorithm into a multilevel nested iteration, and examine how well this approach performs by checking how much work is actually done by evaluating the total usage of work unit (WU), the cost of one matrix-vector multiplication at the finest level.

One other future direction is to utilize the random-walk/diffusion interpretation of Laplace learning and Poisson learning discussed in [34], and develop a novel diffusion-based metric that intrinsically encodes functional label information. We believe that pushing the functional labels into the diffusion process will results in a complete end-to-end utilization of all the information provided in the networks.

Bibliography

[1] D. Achlioptas, *Database-friendly random projections: Johnson-Lindenstrauss with binary coins*, Journal of computer and System Sciences, 66 (2003), pp. 671–687.

[2] M. Ainsworth and J.T. Oden, *A posteriori error estimation in finite element analysis*, Computer Methods in Applied Mechanics and Engineering, 142 (1997), pp. 1 – 88.

[3] ——, *A posteriori error estimation in finite element analysis*, Computer Methods in Applied Mechanics and Engineering, 142 (1997), pp. 1–88.

[4] E. Alpaydin, *Introduction to machine learning*, MIT press, 2020.

[5] O. Axelsson, *Iterative solution methods*, Cambridge university press, 1996.

[6] I. Babuška and W.C. Rheinboldt, *A posteriori error estimates for the finite element method*, International Journal for Numerical Methods in Engineering, 12 (1978), pp. 1597–1615.

[7] R.E. Bank, *Hierarchical bases and the finite element method*, Acta Numerica, 5 (1996), pp. 1–43.

[8] R.E. Bank and R. K. Smith, *A posteriori error estimates based on hierarchical bases*, SIAM Journal on Numerical Analysis, 30 (1993), pp. 921–935.

[9] R.E. Bank and A. Weiser, *Some a posteriori error estimators for elliptic partial differential equations*, Mathematics of Computation, 44 (1985), pp. 283–301.

[10] M. Belkin and P. Niyogi, *Laplacian eigenmaps and spectral techniques for embedding and clustering*, Advances in Neural Information Processing Systems, 14 (2001), pp. 585–591.

[11] ——, *Using manifold structure for partially labeled classification*, Advances in neural information processing systems, 15 (2002).

[12] M. Belkin, P. Niyogi, and V. Sindhwani, *Manifold regularization: A geometric framework for learning from labeled and unlabeled examples.*, Journal of Machine Learning Research, 7 (2006).

[13] M. Benzi, *Preconditioning techniques for large linear systems: a survey*, Journal of Computational Physics, 182 (2002), pp. 418–477.

[14] G.F. Berriz, J.E. Beaver, C. Cenik, M. Tasan, and F.P. Roth, *Next generation software for functional trend analysis*, Bioinformatics, 25 (2009), pp. 3043–3044.

[15] E. Boehnlein, P. Chin, A. Sinha, and L. Lu, *Computing diffusion state distance using Green's function and heat kernel on graphs*, in International Workshop on Algorithms and Models for the Web-Graph, Springer, 2014, pp. 79–95.

[16] B. Bollobás, *Modern graph theory*, vol. 184, Springer Science & Business Media, 2013.

[17] M. Bolten, S. Friedhoff, A. Frommer, M. Heming, and K. Kahl, *Algebraic multigrid methods for Laplacians of graphs*, Linear Algebra and its Applications, 434 (2011), pp. 2225–2243.

[18] S.P. Borgatti, A. Mehra, D.J. Brass, and G. Labianca, *Network analysis in the social sciences*, Science, 323 (2009), pp. 892–895.

[19] D. Braess and J. Schöberl, *Equilibrated residual error estimator for edge elements*, Mathematics of Computation, 77 (2008), pp. 651–672.

[20] A. Brandt, J. Brannick, K. Kahl, and I. Livshits, *Bootstrap algebraic multigrid: status report, open problems, and outlook*, Numerical Mathematics. Theory, Methods and Applications, 8 (2015), pp. 112–135.

[21] A. Brandt, J. J. Brannick, K. Kahl, and L. Livshits, *Bootstrap AMG*, SIAM Journal on Scientific Computing, 33 (2011), pp. 612–632.

[22] A. Brandt, S MacCormick, and J. Ruge, *Algebraic multigrid (AMG) for automatic multigrid solutions with application to geodetic computations, report*, Institute for Computational Studies, Fort Collins, CO, (1982).

[23] J. Brannick, Y. Chen, J. Kraus, and L. Zikatanov, *An algebraic multigrid method based on matching in graphs*, in Domain Decomposition Methods in Science and Engineering XX, Springer, 2013, pp. 143–150.

[24] ——, *Algebraic multilevel preconditioners for the graph Laplacian based on matching in graphs*, SIAM Journal on Numerical Analysis, 51 (2013), pp. 1805–1827.

[25] M. Brezina, R Falgout, S. MacLachlan, T Manteuffel, S McCormick, and J Ruge, *Adaptive smoothed aggregation (αSA)*, SIAM Journal on Scientific Computing, 25 (2004), pp. 1896–1920.

[26] M. Brezina, R. Falgout, S. Maclachlan, T. Manteuffel, S. Mccormick, and J. Ruge, *Adaptive smoothed aggregation (αSA) multigrid*, SIAM Review, 47 (2005), pp. 317–346.

[27] M. Brezina, R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick, and J. Ruge, *Adaptive algebraic multigrid*, SIAM Journal on Scientific Computing, 27 (2006), pp. 1261–1286.

[28] M. Brezina, T. Manteuffel, S. MCormick, J. Ruge, and G. Sanders, *Towards adaptive smoothed aggregation (αSA) for nonsymmetric problems*, SIAM Journal on Scientific Computing, 32 (2010), pp. 14–39.

[29] W.L. Briggs, V.E. Henson, and S.F. McCormick, *A Multigrid Tutorial: Second Edition*, SIAM, 2000.

[30] A. Brückner, C. Polge, N. Lentze, D. Auerbach, and U. Schlattner, *Yeast two-hybrid, a powerful tool for systems biology*, International Journal of Molecular Sciences, 10 (2009), pp. 2763–2788.

[31] E. Bullmore and O Sporns, *Complex brain networks: graph theoretical analysis of structural and functional systems*, Nature Reviews Neuroscience, 10 (2009), pp. 186–198.

[32] J. Calder, *The game theoretic p-laplacian and semi-supervised learning with few labels*, Nonlinearity, 32 (2018), p. 301.

[33] ——, *Consistency of lipschitz learning with infinite unlabeled data and finite labeled data*, SIAM Journal on Mathematics of Data Science, 1 (2019), pp. 780–812.

[34] J. Calder, B. Cook, M. Thorpe, and D. Slepcev, *Poisson learning: Graph based semi-supervised learning at very low label rates*, in International Conference on Machine Learning, PMLR, 2020, pp. 1306–1316.

[35] J. Calder and D. Slepčev, *Properly-weighted graph Laplacian for semi-supervised learning*, Applied Mathematics & Optimization, 82 (2020), pp. 1111–1159.

[36] M. Cao, C.M. Pietras, X. Feng, K.J. Doroschak, T. Schaffner, J. Park, H. Zhang, L.J. Cowen, and B.J. Hescott, *New directions for diffusion-based network prediction of protein function: incorporating pathways with confidence*, Bioinformatics, 30 (2014), pp. i219–i227.

[37] M. Cao, C. M. Pietras, et al., *New directions for diffusion-based prediction of protein function: incorporating pathways with confidence*, Bioinformatics, 30 (2014), pp. i219–i227.

[38] M. Cao, H. Zhang, J. Park, N.M. Daniels, M.E. Crovella, L.J. Cowen, and B. Hescott, *Going the distance for protein function prediction: a new distance metric for protein interaction networks*, PloS one, 8 (2013), p. e76339.

[39] C. Carstensen and S.A. Funken, *Fully reliable localized error control in the fem*, SIAM Journal on Scientific Computing, 21 (1999), pp. 1465–1484.

[40] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien, *Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]*, IEEE Transactions on Neural Networks, 20 (2009), pp. 542–542.

[41] L. Chen, X. Hu, and S. Wise, *Convergence analysis of the fast subspace descent method for convex optimization problems*, Mathematics of Computation, 89 (2020), pp. 2249–2282.

[42] H. Cho, B. Berger, and J. Peng, *Compact integration of multi-network topology for functional analysis of genes*, Cell Systems, 3 (2016), pp. 540–548.

[43] S. Choobdar, M.E. Ahsen, J. Crawford, M. Tomasoni, T. Fang, D. Lamparter, J. Lin, B. Hescott, X. Hu, J. Mercer, et al., *Assessment of network module identification across complex diseases*, Nature Methods, 16 (2019), pp. 843–852.

[44] R.R. Coifman and S. Lafon, *Diffusion maps*, Applied and Computational Harmonic analysis, 21 (2006), pp. 5–30.

[45] C. Colley, J. Lin, X. Hu, and S. Aeron, *Algebraic multigrid for least squares problems on graphs with applications to hodgerank*, in 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2017, pp. 627–636.

[46] Gene Ontology Consortium, *The gene ontology resource: 20 years and still GOing strong*, Nucleic Acids Research, 47 (2019), pp. D330–D338.

[47] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to algorithms*, MIT Press, 2009.

[48] L. Cowen, K. Devkota, X. Hu, J.M. Murphy, and K. Wu, *Diffusion state distances: Multitemporal analysis, fast algorithms, and applications to biological networks*, SIAM Journal on Mathematics of Data Science, 3 (2021), pp. 142–170.

[49] L. Cowen, T. Ideker, B.J. Raphael, and R. Sharan, *Network propagation: a universal amplifier of genetic associations*, Nature Reviews Genetics, 18 (2017), p. 551.

[50] P. D'Ambra and P.S. Vassilevski, *Adaptive AMG with coarsening based on compatible weighted matching*, Computing and Visualization in Science, 16 (2013), pp. 59–76.

[51] T.A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Transactions on Mathematical Software, 38 (2011).

[52] T.A. Davis, S. Rajamanickam, and W.M. Sid-Lakhdar, *A survey of direct methods for sparse linear systems*, Acta Numerica, 25 (2016), pp. 383–566.

[53] P. Destuynder and B. Métivet, *Explicit error bounds in a conforming finite element method*, Mathematics of Computation, 68 (1999), pp. 1379–1396.

[54] K. Devkota, J.M. Murphy, and L.J. Cowen, *GLIDE: combining local methods and diffusion state embeddings to predict missing interactions in biological networks*, Bioinformatics, 36 (2020), pp. i464–i473.

[55] A. El Alaoui, X. Cheng, A. Ramdas, M.J. Wainwright, and M.I. Jordan, *Asymptotic behavior of ℓ_p-based laplacian regularization in semi-supervised learning*, in Conference on Learning Theory, 2016, pp. 879–906.

[56] A. Elmoataz, X. Desquesnes, and O. Lézoray, *Non-local morphological pdes and p-laplacian equation on graphs with applications in image processing and machine learning*, IEEE Journal of Selected Topics in Signal Processing, 6 (2012), pp. 764–779.

[57] A. Elmoataz, F. Lozes, and M. Toutain, *Nonlocal pdes on graphs: From tug-of-war games to unified interpolation on images and point clouds*, Journal of Mathematical Imaging and Vision, 57 (2017), pp. 381–401.

[58] A. Elmoataz, M. Toutain, and D. Tenbrinck, *On the p-laplacian and ∞-laplacian on graphs with applications in image and data processing*, SIAM Journal on Imaging Sciences, 8 (2015), pp. 2412–2451.

[59] T. Fawcett, *An introduction to ROC analysis*, Pattern Recognition Letters, 27 (2006), pp. 861–874.

[60] M. Flores, J. Calder, and G. Lerman, *Analysis and algorithms for ℓ_p-based semi-supervised learning on graphs*, Applied and Computational Harmonic Analysis, 60 (2022), pp. 77–122.

[61] F. Fouss, A. Pirotte, J.M. Renders, and M. Saerens, *Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation*, IEEE Transactions on Knowledge and Data Engineering, 19 (2007), pp. 355–369.

[62] A. Franceschini, D. Szklarczyk, S. Frankild, M. Kuhn, M. Simonovic, A. Roth, J. Lin, P. Minguez, P. Bork, C. vonMering, and L.J. Jensen, *STRING v9.1: protein-protein interaction networks, with increased coverage and integration.*, Nucleic Acids Research, 41 (2013), pp. D808–815.

[63] N. Garcia Trillos, M. Gerlach, M. Hein, and D. Slepčev, *Error estimates for spectral convergence of the graph Laplacian on random geometric graphs toward the Laplace–Beltrami operator*, Foundations of Computational Mathematics, (2019), pp. 1–61.

[64] L. Gómez-Chova, G. Camps-Valls, J. Munoz-Mari, and J. Calpe, *Semisupervised image classification with laplacian support vector machines*, IEEE Geoscience and Remote Sensing Letters, 5 (2008), pp. 336–340.

[65] A. Grover and J. Leskovec, *node2vec: Scalable feature learning for networks*, in Proc. 22nd ACM SIGKDD, ACM, 2016, pp. 855–864.

[66] H. Hakula, M. Neilan, and J.S. Ovall, *A posteriori estimates using auxiliary subspace techniques*, Journal of Scientific Computing, 72 (2017), pp. 97–127.

[67] W.L. Hamilton, R. Ying, and J. Leskovec, *Representation learning on graphs: methods and applications*, arXiv preprint arXiv:1709.05584, (2017).

[68] B.H. Hristov and M. Singh, *Network-based coverage of mutational profiles reveals cancer genes*, Cell systems, 5 (2017), pp. 221–229.

[69] X. Hu, J. Lin, and L.T. Zikatanov, *An adaptive multigrid method based on path cover*, SIAM Journal on Scientific Computing, 41 (2018), pp. S220–S241.

[70] N. Trinajstić I. Gutman, *Graph theory and molecular orbitals. total ϕ-electron energy of alternant hydrocarbons*, Chemical Physics Letters, 17 (1972), pp. 535 – 538.

[71] E. Isaacson and H.B. Keller, *Analysis of numerical methods*, Courier Corporation, 2012.

[72] W.B. Johnson and J. Lindenstrauss, *Extensions of Lipschitz mappings into a Hilbert space*, Contemporary Mathematics, 26 (1984), p. 1.

[73] T. Kavitha, C. Liebchen, K. Mehlhorn, D. Michail, R. Rizzi, T. Ueckerdt, and K. Zweig, *Cycle bases in graphs characterization, algorithms, complexity, and applications*, Computer Science Review, 3 (2009), pp. 199 – 243.

[74] D. W. Kelly, J. P. De S. R. Gago, O. C. Zienkiewicz, and I. Babuška, *A posteriori error analysis and adaptive processes in the finite element method: part I-error analysis*, International Journal for Numerical Methods in Engineering, 19 (1983), pp. 1593–1619.

[75] J.A. Kelner, L. Orecchia, A. Sidford, and Z.A. Zhu, *A simple, combinatorial algorithm for solving sdd systems in nearly-linear time*, in Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing, 2013, pp. 911–920.

[76] J.G. Kemeny and J.L. Snell, *Finite Markov chains: with a new appendix" Generalization of a fundamental matrix"*, Springer, 1976.

[77] J.G. Kemeny, J.L. Snell, and A.W. Knapp, *Denumerable Markov chains: with a chapter of Markov random fields by David Griffeath*, vol. 40, Springer Science & Business Media, 2012.

[78] H. Kim, J. Xu, and L. Zikatanov, *A multigrid method based on graph matching for convection-diffusion equations*, Numerical Linear Algebra with Applications, 10 (2003), pp. 181–195.

[79] L. Koenigsberger, *Hermann von helmholtz*, (1902), p. 357.

[80] S. Kohler, S. Bauer, D. Horn, and P. N. Robinson, *Walking the interactome for prioritization of candidate disease genes*, The American Journal of Human Genetics, 82 (2008), pp. 949–958.

[81] I. Koutis, G. L. Miller, and D. Tolliver, *Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing*, Computer Vision and Image Understanding, 115 (2011), pp. 1638–1646.

[82] J. K. Kraus and S. K. Tomar, *Algebraic multilevel iteration method for lowest order Raviart-Thomas space and applications*, International Journal for Numerical Methods in Engineering, 86 (2011), pp. 1175–1196.

[83] D. Krishnan, R. Fattal, and R. Szeliski, *Efficient preconditioning of Laplacian matrices for computer graphics*, ACM Transactions on Graphics, 32 (2013).

[84] R.B. Lehoucq, D.C. Sorensen, and C. Yang, *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*, vol. 6, SIAM, 1998.

[85] T. Li, R. Wernersson, R.B. Hansen, H. Horn, J. Mercer, G. Slodkowicz, C.T. Workman, O. Rigina, K. Rapacki, H.H. Stærfeldt, S. Brunak, T.S. Jensen, and K. Lage, *A scored human protein–protein interaction network to catalyze genomic interpretation*, Nature Methods, 14 (2017), p. 61.

[86] Y. Li and L.T. Zikatanov, *Nodal auxiliary a posteriori error estimates*, arXiv preprint arXiv:2010.06774, (2020).

[87] Y. Li and L. Zikatanov, *A posteriori error estimates of finite element methods by preconditioning*, Computers & Mathematics with Applications, 91 (2021), pp. 192–201.

[88] C. Liao, K. Lu, M. Baym, R. Singh, and B. Berger, *IsoRankN: spectral methods for global alignment of multiple protein networks*, Bioinformatics, 25 (2009), pp. i253–i258.

[89] D. Liben-Nowell and J. Kleinberg, *The link-prediction problem for social networks*, Journal of the American Society for Information Science and Technology, 58 (2007), pp. 1019–1031.

[90] J. Lin, L.J. Cowen, B. Hescott, and X. Hu, *Computing the diffusion state distance on graphs via algebraic multigrid and random projections*, Numerical Linear Algebra with Applications, 25 (2018), p. e2156.

[91] W. Liu, Y. Jiang, J. Luo, and S. Chang, *Noise resistant graph ranking for improved web image search*, in Conference on Computer Vision and Pattern Recognition, IEEE, 2011, pp. 849–856.

[92] O.E. Livne and A. Brandt, *Lean algebraic multigrid (LAMG): Fast graph Laplacian linear solver*, SIAM Journal on Scientific Computing, 34 (2012), pp. B499–B522.

[93] R. Luce and B.I. Wohlmuth, *A local a posteriori error estimator based on equilibrated fluxes*, SIAM Journal on Numerical Analysis, 42 (2004), pp. 1394–1414.

[94] J. Ma, W. Huang, S. Segarra, and A. Ribeiro, *Diffusion filtering of graph signals and its use in recommendation systems*, in 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2016, pp. 4563–4567.

[95] S. P. MacLachlan, J. D. Moulton, and T. P. Chartier, *Robust and adaptive multigrid methods: comparing structured and algebraic approaches*, Numerical Linear Algebra with Applications, 19 (2012), pp. 389–413.

[96] N. Madras, *A note on diffusion state distance*, arXiv preprint arXiv:1502.07315, (2015).

[97] M. Maggioni and J.M. Murphy, *Learning by unsupervised nonlinear diffusion*, The Journal of Machine Learning Research, 20 (2019), pp. 1–56.

[98] R. Merris, *Laplacian matrices of graphs: a survey*, Linear Algebra and its Applications, 197-198 (1994), pp. 143 – 176.

[99] P. Morin, R. Nochetto, and K. Siebert, *Local problems on stars: a posteriori error estimators, convergence, and performance*, Mathematics of Computation, 72 (2003), pp. 1067–1097.

[100] B. Nadler, N. Srebro, and X. Zhou, *Semi-supervised learning with the graph Laplacian: The limit of infinite unlabelled data*, Advances in Neural Information Processing Systems, 22 (2009), pp. 1330–1338.

[101] A. Nägel, R.D. Falgout, and G. Wittum, *Filtering algebraic multigrid and adaptive strategies*, Computing and Visualization in Science, 11 (2008), pp. 159–167.

[102] A. Napov and Y. Notay, *An algebraic multigrid method with guaranteed convergence rate*, SIAM Journal on Scientific Computing, 34 (2012), pp. A1079–A1109.

[103] ——, *An efficient multigrid method for graph laplacian systems*, Electronic Transactions on Numerical Analysis, 45 (2016), pp. 201–218.

[104] Mark Newman, *Networks*, Oxford University Press, 2018.

[105] A.Y. Ng, M.I. Jordan, and Y. Weiss, *On spectral clustering: Analysis and an algorithm*, in Advances in Neural Information Processing Systems, 2002, pp. 849–856.

[106] F. Nie, X. Wang, M.I. Jordan, and H. Huang, *The constrained laplacian rank algorithm for graph-based clustering*, in Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI Press, 2016, p. 1969–1976.

[107] R.H. Nochetto and A. Veeser, *Primer of Adaptive Finite Element Methods*, Springer, 2012, pp. 125–225.

[108] Y. Notay, *An aggregation-based algebraic multigrid method*, Electronic Transactions on Numerical Analysis, 37 (2010), pp. 123–146.

[109] L. Page, S. Brin, R. Motwani, and T. Winograd, *The pagerank citation ranking: bringing order to the web*, tech. report, Stanford InfoLab, 1999.

[110] W. PRAGER and J. L. SYNGE, *Approximations in elasticity based on the concept of function space*, Quarterly of Applied Mathematics, 5 (1947), pp. 241–269.

[111] S.I. Repin and S.K. Tomar, *Guaranteed and robust error bounds for nonconforming approximations of elliptic problems*, IMA Journal of Numerical Analysis, 31 (2010), pp. 597–615.

[112] M.F. Rios, J. Calder, and G. Lerman, *Algorithms for ℓ_p-based semi-supervised learning on graphs*, arXiv preprint arXiv:1901.05031, (2019).

[113] D.J. Rose, R.E. Tarjan, and G. S. Lueker, *Algorithmic aspects of vertex elimination on graphs*, SIAM Journal on Computing, 5 (1976), pp. 266–283.

[114] G. Rucker, *Network meta-analysis, electrical networks and graph theory*, Research Synthesis Methods, 3 (2012), pp. 312–324.

[115] A. Ruepp, A. Zollner, D. Maier, K. Albermann, J. Hani, M. Mokrejs, I. Tetko, U. Guldener, G. Mannhaupt, M. Munsterkotter, and H. W. Mewes, *The FunCat, a functional annotation scheme for systematic classification of proteins from whole genomes*, Nucleic Acids Research, 32 (2004), pp. 5529–5545.

[116] Y. Saad and M.H. Schultz, *Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 7 (1986), pp. 856–869.

[117] J.R. Shewchuk et al., *An introduction to the conjugate gradient method without the agonizing pain*, 1994.

[118] J. Shi and J. Malik, *Normalized cuts and image segmentation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 22 (2000), pp. 888–905.

[119] Z. Shi, S. Osher, and W. Zhu, *Weighted nonlocal laplacian on interpolation from sparse data*, Journal of Scientific Computing, 73 (2017), pp. 1164–1177.

[120] D. Slepcev and M. Thorpe, *Analysis of p-laplacian regularization in semisupervised learning*, SIAM Journal on Mathematical Analysis, 51 (2019), pp. 2085–2120.

[121] Z. Song, X. Yang, Z. Xu, and I. King, *Graph-based semi-supervised learning: A comprehensive review*, IEEE Transactions on Neural Networks and Learning Systems, (2022).

[122] D. Szklarczyk, A. Franceschini, S. Wyder, K. Forslund, D. Heller, J. Huerta-Cepas, M. Simonovic, A. Roth, A. Santos, K.P. Tsafou, et al., *String v10: protein–protein interaction networks, integrated over the tree of life*, Nucleic Acids Research, 43 (2014), pp. D447–D452.

[123] D. Szklarczyk, A.L. Gable, D. Lyon, A. Junge, S. Wyder, J. Huerta-Cepas, M. Simonovic, N.T. Doncheva, J.H. Morris, P. Bork, L.K. Jensen, and C.V. Mering, *String v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets*, Nucleic acids research, 47 (2019), pp. D607–D613.

[124] X. Tai and J. Xu, *Global and uniform convergence of subspace correction methods for some convex optimization problems*, Mathematics of Computation, 71 (2002), p. 105–124.

[125] S.K. Tomar and S.I. Repin, *Efficient computable error bounds for discontinuous Galerkin approximations of elliptic problems*, Journal of Computational and Applied Mathematics, 226 (2009), pp. 358 – 369.

[126] J. C. Urschel, J. Xu, Hu X., and L. T. Zikatanov, *A cascadic multigrid algorithm for computing the fiedler vector of graph laplacians*, Journal of Computational Mathematics, 33 (2015), pp. 209–226.

[127] R. Verfürth, *A posteriori error estimates for nonlinear problems. finite element discretizations of elliptic equations*, Mathematics of Computation, 62 (1994), pp. 445–475.

[128] P.S. Vassilevski and L.T. Zikatanov, *Commuting projections on graphs*, Numerical Linear Algebra with Applications, 21 (2014), pp. 297–315.

[129] R. Verfürth, *A posteriori error estimation techniques for finite element methods*, OUP Oxford, 2013.

[130] M. Wang, H. Li, D. Tao, K. Lu, and X. Wu, *Multimodal graph-based reranking for web image search*, IEEE Transactions on Image Processing, 21 (2012), pp. 4649–4661.

[131] K.Q. Weinberger, F. Sha, Q. Zhu, and L.K. Saul, *Graph Laplacian regularization for large-scale semidefinite programming*, in Proceedings of the 19th International Conference on Neural Information Processing Systems, MIT Press, 2006, p. 1489–1496.

[132] J. Weston, F. Ratle, and R. Collobert, *Deep learning via semi-supervised embedding*, in Proceedings of the 25th international conference on Machine learning, 2008, pp. 1168–1175.

[133] J. Xu, *Iterative methods by space decomposition and subspace correction*, SIAM Review, 34 (1992), pp. 581–613.

[134] J. Xu and L. Zikatanov, *Algebraic multigrid methods*, Acta Numerica, 26 (2017), p. 591–721.

[135] W. Xu and L.T. Zikatanov, *Adaptive aggregation on graphs*, Journal of Computational and Applied Mathematics, 340 (2018), pp. 718 – 730.

[136] Z. Yang, W. Cohen, and R. Salakhudinov, *Revisiting semi-supervised learning with graph embeddings*, in International conference on Machine Learning, PMLR, 2016, pp. 40–48.

[137] G. Yu, G. Fu, J. Wang, and H. Zhu, *Predicting protein function via semantic integration of multiple networks*, IEEE/ACM Transactions on Computational Biology and Bioinformatics, 13 (2015), pp. 220–232.

[138] J. Yu, D. Tao, and M. Wang, *Adaptive hypergraph learning and its application in image classification*, IEEE Transactions on Image Processing, 21 (2012), pp. 3262–3272.

[139] D. Zhou, J. Huang, and B. Schölkopf, *Learning from labeled and unlabeled data on a directed graph*, in Proceedings of the 22nd international conference on Machine Learning, 2005, pp. 1036–1043.

[140] D. Zhou and B. Schölkopf, *Regularization on discrete spaces*, in Pattern Recognition: 27th DAGM Symposium, Vienna, Austria, August 31-September 2, 2005. Proceedings 27, Springer, 2005, pp. 361–368.

[141] X. Zhou and M. Belkin, *Semi-supervised learning by higher order regularization*, in Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, JMLR Workshop and Conference Proceedings, 2011, pp. 892–900.

[142] X. Zhu, *Semi-supervised learning with graphs*, Carnegie Mellon University, 2005.

[143] X. Zhu and J.D. Ghahramani, Z.and Lafferty, *Semi-supervised learning using Gaussian fields and harmonic functions*, in Proceedings of the 20th International conference on Machine Learning (ICML-03), 2003, pp. 912–919.