PREPROCESSING AND LEARNING FOR GRAPH STRUCTURED DATA

Christopher Brissette

Submitted in Partial Fullfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

Approved by:
George Slota, Chair
Boleslaw K. Szymanski
Jianxi Gao
Andy Huang

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York

[August 2023]
Submitted July 2023

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

Writing this thesis and reflecting on the past four years has been an adventure in more ways than one. Obviously it was not my preference to have half of my PhD take place during a pandemic, but I think that helped contribute to a very eclectic body of work to look back at and appreciate. The connections I have made during these years, both at RPI and elsewhere have left distinct marks on this thesis.

First, I need to thank my advisor George Slota. From the moment I stepped foot on the RPI campus, George was been supportive of my interests. Sometimes this may have been to a fault, as not all ideas ended up working out in the end. Even now, somewhere in Amos Eaton, across several white boards and computers, lies a graveyard of half-completed projects and wild ideas. George was always there to offer guidance, throw out new crazy theories, and connect me with researchers. I don't think I have a definitive view of what an advisor should be, but I know George was the advisor I needed.

Secondly I have to thank my committee. I feel incredibly lucky to have a committee I have worked with as closely as I have. Everyone feels more like a coworker than an academic stranger. To Andy Huang, professor Boleslaw Szymanski, and professor Jianxi Gao, thank you for your knowledge, stories, and shared experiences. Also for professors Szymanski and Gao, sorry that none of our papers made it into this thesis. They didn't quite match the tenuous theme!

My family deserves a big thanks as well. My mother, Lisa, and father, Aaron, have been actively confused and supportive of my pursuits for 10 years now. They gladly put my papers on the fridge as though they are macaroni art, and are still convinced that I can fix their computer problems because I studied computer science. I would never have gotten this far if not for the love and support they both provided me as a child and continue to provide to this day. My father always said, "*If you get out of school before you are thirty, you did something wrong.*" Well, I got close. I hope I don't regret leaving those last two years on the table.

Finally, my partner Bethany deserves all the thanks in the world. When we started dating at the beginning of college 10 years ago, I made it very clear I wanted a PhD. I don't think either of us knew what 10 years would look like then. But here we are. Through late nights, solo travel, and a lot of self doubt, she is still there. She has been patient,

supportive, excited, and loving through the whole ordeal. As I stand at the end of my academic experience, there is one person who I have spent more time beside than anyone else, and there is no single person I would prefer.

Actually I lied. I need to thank my *true* muse. Thanks to the funding sources that made this all possible! This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) Program through the FASTMath Institute under Contract No. DE-SC0021285 at the Rensselaer Polytechnic Institute, Troy NY. This work was also supported under Laboratory Directed Research and Development (LDRD) program, Project No. 218474, at Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

ABSTRACT

Graphs are general structures which may be used to describe any system or dataset with related elements. Because of the prevalence of such data, efficient and accurate algorithms for analyzing graphs are of extreme importance in numerical algorithms and general data science tasks. The size of graph structured datasets has only increased in the past decades and promises to continue doing so. As companies like Google and Meta wrestle with peta and exa-scale graph analysis problems: computational scientists face many of the same issues as simulations require ever-larger meshes. Because of this, acceleration and preprocessing techniques are important to ensure graph algorithms run efficiently and accurately. We investigate several preprocessing and acceleration techniques for performing tasks on graph structured data.

We develop a methodology for generating graph null-models with a desired degree distribution. This is a problem which has been of interest to network scientists for decades. Despite this, parallelizable, fast subroutines used in algorithms for generating such graphs tend to yield inaccurate distributions. We suggest a novel analysis technique for the popular Chung-Lu random graph generator, and show that this analysis technique provides a method for automatically generating parameters for Chung-Lu-like null models as a pre-processing step. We provide several methods for generating these null-models, and show that in all cases we significantly out-perform standard Chung-Lu generation. We also suggest that such null models may be used to improve the accuracy of Modularity maximization.

Additionally, we examine the task of coarsening graphs while preserving the spectrum of the graph Laplacian. Coarsening is an important preprocessing step for many large scale graph problems which aim to solve relatively small sub-problems and reconstruct an approximate solution on the original graph. Coarsening is used in clustering, partitioning, and multigrid methods for solving linear systems of equations. The graph Laplacian is an important operator for describing graph structured data. It relates the heat transfer in a graph to its topology. As such, its eigenvectors and eigenvalues hold important information about edge cuts and clustering. We present a heuristic for preserving the spectrum of the graph Laplacian during coarsening, and present a parallel algorithm for utilising this heuristic. This is in contrast to prior publications on the subject which focus on serial and k-means methods for spectrum consistent coarsening. We further analyze the inverse problem, and

find that the original graph may be reconstructed to within some edge-weight error given a coarse representation which approximates its spectrum. This presents a novel development in graph coarsening literature and suggests that preserving a graphs spectrum during coarsening may be sufficient to preserve all structure.

Finally we investigate a technique for accelerating the training of graph neural networks using Koopman operator theory. Graph neural networks provide a powerful method for performing classification and prediction tasks on graphs. This is in contrast to traditional neural networks which struggle with the unordered nature of nodes and edges. Because of this, a great deal of effort has been put into accelerating graph neural networks through techniques such as graph pooling. Despite this, they are still often slow to train and have significant memory overheads. We suggest a method for accelerating training by interweaving standard backpropagation steps with prediction steps that make use of simple matrix-vector multiplication. We apply our method to the task of node classification and find that it is prone to instability, but can achieve multiple times speed-ups over Adam for well-chosen parameters. This work represents the first time Koopman training has been applied to graph neural networks, and the first time it has been applied on GPU.

CHAPTER 1
INTRODUCTION

1.1 Overview

This thesis concerns itself with studying techniques for preprocessing graph data, and performing learning tasks on graphs with the intent of improving algorithmic metrics such as accuracy and run-time. Graphs allow for a remarkably general description for many problems, and have been applied in settings such as quantum mechanics [1], social science [2], [3], and numerical PDE [4]. Beyond this, they are of fundamental importance to computer science and the study of algorithms [5]. Graphs present a framework for studying objects based on their relational properties. In the language of graph theory, these objects are represented as *vertices*, sometimes called *nodes*, and their relationships are referred to as *edges*. A graph is then a collection of nodes and edges $G = (V, E)$, where V represents the set of nodes, and E is the set of edges

More specifically, this thesis focuses on three distinct graph problems, null graph generation, graph coarsening, and node classification. While not explicitly related, each of these problems are closely tied to graph clustering in ways that will be made clear within the later text. Broadly speaking, graph clustering is the task of grouping the vertices of a graph with respect to some metric. Most often this means approximately minimizing the number of edges between clusters.

1.2 Null graph generation

Generating graphs with given properties has a significant history. While preserving graph properties such as the spectrum have been studied somewhat, the problem of generating graphs with a given degree sequence has received an overwhelming amount of research. The degree sequence of a graph defines many important properties such as the epidemic threshold for disease models [6] and the resilience [7], [8] of a network. Due to its link with important graph metrics, generating graphs with given degree distributions is an important problem

for graph null-hypothesis testing [9], [10]. Understanding if certain observed properties of a graph are unique to an individual instance, or just arise as a consequence of its degree distribution is important in many applications such as graph clustering [11], [12]. In this thesis, Chung-Lu random graph generation [13] is focused on. This method is mathematically simple, and as such is the basis for the theoretical analysis of many graphs as well as the modularity metric which is used in clustering [14]. Despite this, Chung-Lu graph generation gives rise to significant degree sequence errors when the underlying sequence does not have certain properties.

1.3 Graph coarsening

Graph coarsening is a pre-processing technique where some graph $G = (V, E)$ is reduced in size to the graph $G_c = (V_c, E_c)$ such that $|V_c| < |V|$. The goal of such a process is to solve some problem on G_c which may otherwise be intractable on G, and then "map" this solution back to the original graph. This yields an approximate solution to the problem which is either sufficient for the given application, or can be further refined. The methods used for graph coarsening range from expensive spectral methods to greedy metric maximization and it is widely used in numerical PDE and clustering applications. Of some recent interest is the coarsening of graphs while preserving spectral properties of various operators such as the normalized Laplacian. This is largely because it has been shown that these methods may provide significant speedups for training graph neural networks (GNN). Comparatively little effort has been spent on the inverse problem. Given a coarsened graph, whose operator spectrum approximates some original graph operator, how accurately can the topology of the original graph be recovered. This can be seen as quantifying how well a given coarsening preserves structure, and provides an alternate viewpoint for why such a coarsening may be applied to training graph neural networks.

1.4 Node classification

The problem of classifying nodes in a graph based on a graphs topology and a set of features assigned to each node is a learning problem with applications to protein protein interaction networks, social network analysis, and clustering. Traditionally, learning methods have

struggled to perform this task since graph data does not inherently have a canonical ordering. For this purpose a great deal of effort has been invested in creating methods for computing graph embeddings. In 2017, the graph convolution network (GCN) was introduced by Kipf and Welling which provided a method for learning from graph data with impressive results. Despite this, the method was limited due to its scalability, and many methods have arisen to deal with this. These methods include graph sparsification, graph coarsening, and graph pooling. Each of these methods aim to downsize the data to reduce the cost of the message-passing and backpropagation steps. Alternatively, the training time may also be reduced via optimizer improvements. Recently it has been shown that Koopman operators, which approximate the evolution of nonlinear dynamics based on observed states, may provide improvements for optimizers on deep neural networks. Until now these methods have lacked practicality as they have not been implemented on GPU. Furthermore their accuracy has not been demonstrated on GCN problems.

1.5 Organization

The remainder of this thesis is organized into three chapters detailing four first author papers as well as various workshop papers and presentations. Chapter 2 discusses novel analysis methods for predicting the output degree sequences of Chung-Lu random graph generation for simple graphs. The chapter also discusses various algorithmic extensions for improving the accuracy of these sequences in an L1-sense. The papers which constitute this section are **Limitations of Chung-Lu Random Graph Generation**: *published in CNA21*, and **Correcting Output Degree Sequences in Chung-Lu Random Graph Generation**: *published in CNA22*.

Chapter 3 discusses the inverse problem of spectral graph coarsening. In this section, theoretical bounds are derived for the absolute difference between the adjacency matrix of an original graph $G = (V, E)$ and a "recovered" graph $\hat{G}, (\hat{V}, \hat{E})$, given a spectral approximation bound. Additionally a method for coarsening graphs while preserving the spectrum of the graph Laplacian is presented. This work follows the paper **Spectrum Consistent Coarsening Approximates Edge Weights**: *set to appear in SIMAX*. Additionally this chapter discusses implementation details for parallel-coarsening which were presented in the paper **Parallel Coarsening of Graph Data with Spectral Guarantees**: *published in*

the proceedings of the SIAM TDA Workshop 2022.

Chapter 4 focuses on the use of Koopman operators for accelerating the training of node classification on GPU. This section concerns itself with the manuscript **Acceleration of GNN Node Classification Using Koopman Operator Theory on GPU**: *currently under review.*

CHAPTER 2
RANDOM GRAPH GENERATION

2.1 Chapter overview

In this chapter we focus on analyzing and correcting sequence errors of graph null models. We specifically investigate Chung-Lu random graph generation [13], as it is an easily scaled method, amenable to significant parallelization. There are known degree sequence properties which guarantee the accuracy of Chung-Lu generation, however these are incredibly restrictive in the case of simple graphs where multi-edges are disallowed. We present a matrix formalism for predicting the output of Chung-Lu random graph generation and show that in some cases, improving the degree sequence output can be reduced to solving a linear system. Perturbations of this matrix formalism are then used to improve the accuracy of Chung-Lu graph generation with the aid of standard learning techniques.

2.2 Modeling Chung-Lu

2.2.1 Introduction to modeling Chung-Lu

Say we wish to generate a random simple graph $G = (V, E)$ with a degree distribution $y = \{N_1, N_2, \cdots, N_m\}$ where N_k represents the numbers of nodes with degree k. Here, simple means that there are no self-loops or multi-edges. This is a problem that arises in many network science applications, most notably for the generation of null-models used for basic graph analytics [15]. Generating such simple networks exactly using the explicit configuration model, or erased configuration model is computationally expensive for even moderately large networks [9]. This is in part because the explicit configuration model is difficult to parallelize, and partly because correcting self-loops and multi-edges in the

Portions of this chapter previously appeared as: C. Brissette and G. M. Slota, "Limitations of chung lu random graph generation," in *Proc. Tenth Int. Conf. Complex Netw. Appl.*, 2021, pp. 451–462.

Portions of this chapter previously appeared as: C. Brissette, , D. Liu, and G. M. Slota. "Correcting output degree sequences in Chung-Lu random graph generation," *Proc. Eleventh Int. Conf. Complex Net. Appl.*, 2022, pp.69-80.

erased configuration model can be cumbersome. As such, we rely on probabilistic methods for large-scale graph generation that only match y in expectation. The Chung-Lu random graph model [13] is one such widely-used probabilistic model. This model pre-assigns to each node $v_i \in V(G)$ a weight w_i corresponding the the degree we wish for the node to have. It then connects all nodes v_i, v_j pairwise with the probability $p_{ij} = \frac{w_i w_j}{\sum w_k}$. It is known that the degree of each node in the output graph will match its pre-assigned weight in expectation. There are a number of ways that generating such graphs can be done computationally. Some methods generate loops and multi-graphs, while others generate simple graphs. We focus on what is sometimes called the Bernoulli Method for generating Chung-Lu graphs [16], as it is amenable to the edge-skipping technique [17] that allows linear work complexity and near-constant parallel time for scalable implementations [10], [18], [19]. In this method, we implicitly consider all possible pairs of edges between unique nodes and generate edge (v_i, v_j) with $i \neq j$ according to the probability p_{ij}. This generates a simple-graph with degree sequence $\tilde{y}$ where $\mathbb{E}[\tilde{y}] = y$. While we focus our analysis on this specific variation, as it is the one most likely to be used in practice, other methods that generate multi-edges and/or loops have many of the same issues that we discuss in this chapter.

The Chung-Lu model, though popular and theoretically sound under the tame condition that $w_i w_j < \sum_{k=1}^{m} w_k$ for all $v_i, v_j \in V$, can produce degree distributions drastically different from the desired expectation in practical settings. This has been widely noted and addressed in the literature [10], [16], [20]–[24]. To theoretically motivate why this is the case, consider generating a graph that has many nodes of degree two. While Chung-Lu guarantees that the expected degree of each of these nodes will be two, it makes no other guarantees regarding the probability mass function of these degrees. Indeed in practice, nodes with expected degree two often have degree 0, 1, 3, and beyond. This is particularly challenging when Chung-Lu generation is utilized as a subroutine for more complex graph generation, such as when generating graphs that also match a clustering coefficient distribution (e.g., the BTER model) [25] or a community size distribution for community detection benchmarking [19], [26]. In these and other instances, minimizing error in the degree distribution is critical, as this error can propagate through the rest of the generation stages. In Figure 2.1 we see an example of the observed error when generating some graphs. As can be seen, the output of Chung-Lu in both cases underestimates the number of degree one nodes, and accrues additional error from other low degree families as well. This suggests that instead

of strictly caring about the expected degree of each node in Chung-Lu generation, as is generally done, we should additionally consider deeper statistical properties of the model in application.

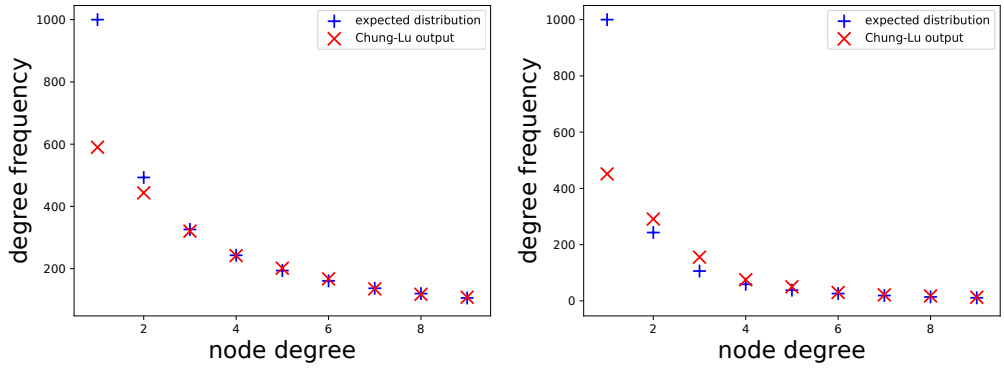


Figure 2.1: Distribution error of Chung-Lu. We consider the degree classes between one and nine for two different power law distributions. On the left is a power-law distribution with exponent $\beta = 1.0$ and on the right is a power-law distribution with exponent $\beta = 2.0$. In the top two plots, crosses represent the input distribution and x's represent the average distribution for 20 instances of Chung-Lu graphs given the power law distribution distribution as input. We can see that the Chung-Lu generated graphs drastically under-represent degree one nodes. This is a phenomenon that commonly occurs in application and can greatly affect generation accuracy.

To better understand the output of Chung-Lu, consider grouping all nodes by expected degree. That is, take degree families $d_k = \{v_i \in V : w_i = k\}$ and consider connections between them. From the point of view of a single node $v_j \in V$ with expected degree w_j the number of connections it has to each degree family d_k is binomially distributed with mean $\frac{kw_j}{\sum w_i}|d_k|$. Therefore the degree distribution of the nodes in d_{w_j} is the sum of m independent binomial random processes where m is the maximum expected degree of the graph. This allows us to go beyond simply guaranteeing the mean of each degree family, and instead predict the probability mass function for the degrees of each of these families, and by extension predict degree distribution errors.

Since the degree distribution of each degree family d_k in our graph is binomially distributed, we may apply a further approximation. Because the limiting case of the binomial distribution is the Poisson distribution, we approximate the number of connections between nodes in a given degree family with all other nodes as a sum of Poisson distributions, which

is again Poisson. We note that often times a desired degree distribution will be such that certain degree classes will not have the number of nodes required for this approximation to have guaranteed accuracy. In fact, power-law degree distributions will in general have degree families d_k where $k \approx m$ such that $|d_k| \approx 1$. However, we also note that this additionally means the node-wise error contributed by those families is relatively small, so we are willing to sacrifice some accuracy in lieu of a cleaner description.

Say that X_{ij} is the Poisson distribution representing the degree of each node in d_i if d_i only connected to nodes in d_j. Additionally, take the mean of this Poisson distribution to be γ_{ij}. Because the means of independent Poisson distributions are additive, we have the following linear system, describing the means of each distribution.

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1m} \\ \gamma_{21} & \gamma_{22} & \cdots & \gamma_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{m1} & \gamma_{m2} & \cdots & \gamma_{mm} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_m \end{bmatrix} \tag{2.1}$$

This matrix provides additional rationale for our Poisson approximation. Since we assumed the distributions were Poisson we may now add means of Poisson distributions directly as opposed to computing with more complex independent binomial distributions. In the case of the Chung-Lu model, each $\gamma_{ij} = \frac{w_i w_j}{\sum w_k}$. This, perhaps as expected, gives the right hand means of $\mu_k = k$. This means that the degrees within each degree family d_k should be approximately Poisson distributed about k. Before moving on we note that a similar analysis can be done for any connection probabilities. While we are focusing on Chung-Lu probabilities, this model also describes the output degree sequence for any set of chosen p_{ij} between degree families, albeit with potential changes to the means μ_i.

Given the description offered by Equation 2.1 we now have the tools to estimate the output of Chung-Lu through a simple linear system. Consider an input degree distribution $y = [N_1, N_2, \cdots, N_m]^T$ as a vector in $\mathbb{R}^m$ with the number of nodes being, $N = \sum_{i=1}^{m} N_i$. Additionally take $poiss(k)$ to be the probability density function of the Poisson distribution

with mean k. We can calculate the expected output $\tilde{y}$ of this as follows.

$$\mathbf{Q}y = \begin{bmatrix} | & | & & | \\ poiss(1) & poiss(2) & \cdots & poiss(m) \\ | & | & & | \end{bmatrix} \begin{bmatrix} N_1 \\ N_2 \\ \vdots \\ N_m \end{bmatrix} = \tilde{y} \tag{2.2}$$

This construction works in the following way. Each column of our matrix represents the probability mass function of degrees within each degree family. Taking the inner product of a row r of this matrix with a vector of degree family sizes amounts to adding together the expected number of degree r nodes produced by each degree family under the Chung-Lu algorithm. So, by considering the action of the entire matrix, we are considering the action of Chung-Lu as a whole. Note here we are assuming $poiss(k)$ is the full, discrete version of the Poisson distribution with mean k. This implies that the system in Equation 2.2 maps $\mathbb{R}^m$ to an infinite dimensional space. This is not computationally useful. We therefore truncate the Poisson matrix $\mathbf{Q}$ to be square in $\mathbb{R}^{m\times m}$ by removing the first row corresponding to degree zero nodes, as well as everything below the m^{th} row. We will denote this matrix by $\mathbf{P}$. Our justification for this truncation is two-fold. One, we are inputting a degree sequence from $\mathbb{R}^m$, and we mainly only care about error with regards to those output degrees between one and m. Two, making the matrix square allows for us to invert the matrix which will be useful for generating Chung-Lu graphs with more accurate degree sequences. Note that truncating $\mathbf{Q}$ to some dimension m amounts to ignoring nodes with degree zero as well as nodes with degree higher than m. If we wish to obtain error information for higher degrees as well we can easily append zeros to the end of our input degree sequence and consider $\mathbf{P}\in\mathbb{R}^{n\times n}$ where $n > m$ and m is the maximum degree of our desired degree sequence. Then, for large enough n, our error is only ignoring nodes of degree zero. In a practical setting, these nodes would possibly be thrown out and ignored, anyways. The rest of this paper discusses properties of $\mathbf{P}$, the limitations these properties suggest, and how the matrix can be used to improve the accuracy of Chung-Lu outputs in some cases.

2.2.2 Contributions

As noted, while Chung-Lu graph generation is a useful tool for many theoretical purposes and is used widely in fields such as social network analysis, it often does a poor job of approximating degree sequences at the ends. The specific issue considered in this chapter is that Chung-Lu generated networks will often under-represent low degree nodes. In Figure 2.1, we can clearly see that actual Chung-Lu realizations may easily contain less than 60 percent of the desired number of degree one nodes. This can lead to a great deal of inaccuracy for sequences with particularly large numbers of low degree nodes, such as those arising from power-law distributions. In practice, this generally means that generated graphs will have many vertices of degree zero, so one way of resolving this issue is to connect these nodes to the graph in order to inflate the number of degree one nodes. Depending on the degree sequence, this can easily skew other degree classes without careful choice of where these nodes are connected. This may also require considerable computation. For this reason, it is far simpler for applications to throw away degree zero nodes. Other proposed methods might artificially inflate the input sequence in terms of degree one nodes so the output better matches the desired input [25], but this also presents similar challenges.

For this reason, we suggest the matrix model referenced in the introduction. The standard input degree sequence for Chung-Lu is simply the desired output sequence y. We suggest a "shifted" Chung-Lu algorithm where, given a matrix model $\mathbf{P}$ for the output of the Chung-Lu algorithm, we take our desired output sequence y and solve for $x = \mathbf{P}^{-1}y$. Then the input to a Chung-Lu graph generator is x as opposed to the desired output. This is particularly compelling since the matrix $\mathbf{P}^{-1}$ only depends on the maximum degree of our desired output sequence and once computed allows for drastic accuracy improvement at negligible algorithmic cost. While useful in certain special cases, we find that such an algorithm is not possible in general. We prove several the matrix $\mathbf{P}$ is invertable and show that many distributions do not have non-negative inverses. We investigate these cases and classify some instances in which an inverse distribution is guaranteed to have negative entries. Most interestingly, we provide tight bounds on the expected maximum number of nodes that may belong in each degree family for both non-increasing as well as general distributions. These bounds suggest that there exists a vast number of graphs that Chung-Lu generation is ill-equipped to generate.

2.2.3 Properties of the matrix model

From the introduction, we use the assumption that the degree distribution of each family is approximately Poisson distributed to form a matrix that will transform input distributions into approximate output distributions from the Chung-Lu model. Assume that our input distribution has degrees in $\mathbb{N}_m = \{1, \cdots, m\}$ and is represented by $x = [N_1, N_2, \cdots, N_m]^T$ where N_k represents the number of nodes with expected degree k and $N = \sum_{k=1}^{m} N_k$. We can represent our matrix $\mathbf{P}$ in terms of the following factorization.

$$\begin{aligned}\mathbf{P} &= \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & \frac{1}{2!} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{m!} \end{bmatrix} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & m \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 2^{m-1} & \cdots & m^{m-1} \end{bmatrix} \begin{bmatrix} e^{-1} & 0 & \cdots & 0 \\ 0 & 2e^{-2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & me^{-m} \end{bmatrix} \\ &= \mathbf{AVB}\end{aligned} \tag{2.3}$$

When this factorization is multiplied out, we obtain exactly the $\mathbf{P}$ matrix discussed in the introduction. Note that realizing a Chung-Lu graph model amounts to computing $\mathbf{P}x$ for some pre-defined x. We instead look at the inverse problem of determining $x \in \mathbb{R}^{+m}$ given $\mathbf{P} \in \mathbb{R}^{m \times m}$ and desired output $y \in \mathbb{R}^{+m}$. Here $\mathbb{R}^{+m}$ is the element-wise positive region of $\mathbb{R}^m$. This amounts to solving the linear system $\mathbf{P}x = y$. One may be tempted to simply invert this matrix using any number of computational methods, and this is reasonable for small m. However, given the factorization in Equation 2.3, we have that $\mathbf{P} = \mathbf{AVB}$ with $\mathbf{V}$ a Vandermonde matrix. Due to the extremely poor conditioning of both $\mathbf{A}$ and $\mathbf{V}$, using a computational method for inverting $\mathbf{P}$ is not advised. Fortunately $\mathbf{A}$ and $\mathbf{B}$ are diagonal, meaning they are easy to invert, so finding the inverse of $\mathbf{P}$ only requires finding an inverse to $\mathbf{V}$. Again, we do not want to compute this using standard computational methods, since Vandermonde matrices are the textbook examples of nearly uninvertible matrices. Fortunately, our Vandermonde matrix is such that it has a special structure yielding a somewhat simple closed-form inverse given in [27]. It relates each entry in the matrix to associated binomial coefficients and Stirling numbers of the first kind. Explicitly, each entry

is expressed as follows.

$$\mathbf{V}_{ij}^{-1} = (-1)^{i+j} \sum_{k=max(i,j)}^{n} \frac{1}{(k-1)!} \binom{k-1}{i-1} \begin{bmatrix} k \\ j \end{bmatrix} \tag{2.4}$$

For distributions with entry-wise positive inverses we can now compute the input of Chung-Lu that will best approximate the desired output according to $\mathbf{P}^{-1}y = x$. The actual implementation of this would look like the pseudocode given in Algorithm 1 where $\lfloor \cdot \rfloor$ represents element-wise rounding down to the nearest integer.

Algorithm 1 ShiftedChungLu (y,d_{max})

1: S $\leftarrow$ ComputeStirlingMatrix(d_{max})
2: $A^{-1} \leftarrow$ ComputeDiagInverse(A)
3: $B^{-1} \leftarrow$ ComputeDiagInverse(B)
4: $V^{-1} \leftarrow$ ComputeVandermondeInverse(S,d_{max})
5: $\tilde{x} \leftarrow \lfloor B^{-1}V^{-1}A^{-1}y \rfloor$
6: $G \leftarrow$ GenerateChungLu($\tilde{x}$)
7: **return** G

2.2.3.1 Not all solutions are positive

We now concern ourselves with cases where Algorithm 1 will fail. These cases will occur exactly when $\mathbf{P}^{-1}y$ has negative entries. To understand why this is the case, consider that $\mathbf{P}^{-1}y$ represents a degree distribution. Negative entries in this vector therefore represent a meaningless value as an input to the Chung-Lu algorithm. Matrix $\mathbf{P}$ has only positive real entries. This implies that for any element-wise positive vector x, $\mathbf{P}x$ is also positive. While this implies that any positive input will yield an approximately valid result, it does not exclude the possibility of vectors with negative entries also mapping into the positive region of $\mathbb{R}^m$ under the action of $\mathbf{P}$. This means that we may not be able to use the output of $\mathbf{P}^{-1}y = x$ as the input of $\mathbf{P}x$ since x has the possibility of containing negative elements. In Figure 2.2, we can see what the action of $\mathbf{P}$ looks like on a sample of random vectors for $\mathbf{P}\in\mathbb{R}^4$. Notice how, as expected, it "squishes" the positive region into a small sliver.

Given a number of nodes N we look to bound how many nodes of each degree are feasible. That is, if we have some degree distribution x with L1-norm $\|x\|_1 = N$ we wish to find lower and upper bounds, l_i and u_i respectively on $|(\mathbf{P}x)_i|$ such that $l_i \leq |(\mathbf{P}x)_i| \leq u_i$. We want to do this for every degree family. Take the projector $\rho_i = e_i^T e_i$ where e_i is the i^{th}

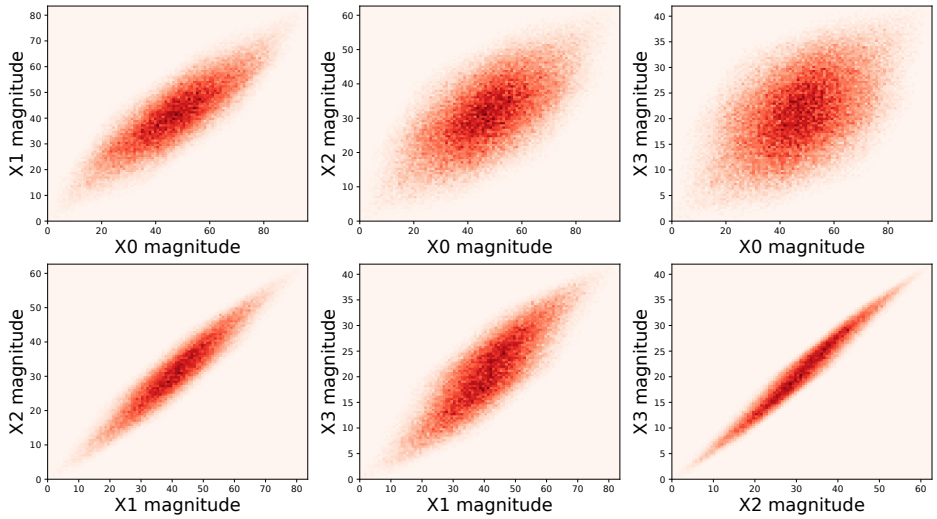


Figure 2.2: Action of P on the positive hypercube. Here we can see plots of projections of random vectors under the action of P as a heat map. The sample consists of 100,000 random vectors with random integer entries selected to be within $\{0, \cdots, 100\}$ under the action of $\mathbf{P} \in \mathbb{R}^{4\times4}$. The output vectors are then projected onto each canonical unit vector $e_j \in \mathbb{R}^4$ and plotted pairwise. These vectors are referred to as Xi in the axis labels. Intuitively this shows all feasible output from a Poisson random graph model with node degrees limited to those in $\{1, 2, 3, 4\}$. We can see that all positive vectors remain inside the positive region as expected, and we also see how sharply limiting this is for finding positive solutions of $\mathbf{P}^{-1}y$ for y positive.

canonical unit vector in $\mathbb{R}^m$. Then we know $|(\mathbf{P}x)_i| = \|\rho_i \mathbf{P}x\|_1$. This directly implies from the structure of $\mathbf{P}$ that we have,

$$N \min_k |\mathbf{P}_{ik}| \leq \|\rho_i \mathbf{P}x\|_1 \leq N\mathbf{P}_{ii} \, \forall \, x \in \mathbb{R}^m \, : \, \|x\|_1 = N \tag{2.5}$$

Under the necessary, but reasonable, assumption that $N > m$, Equation 2.5 gives us a tight upper bound on the number of nodes we can reliably generate of a given degree based on only the number of nodes in our distribution. This bound is realized precisely when all of the nodes in our distribution have input degree $\frac{N}{i}$. We may be interested in what outputs a more narrow space of input distributions can reliably generate. Consider bounding the number of nodes with given degrees in a special case. Namely we pick degree family sizes

such that the following is true.

$$N_1 \geq N_2 \geq \cdots \geq N_m \tag{2.6}$$

That is, the size of the families are non-increasing with respect to input degree. This classifies a wide variety of networks ranging from those with identical family sizes, to power-law distributions. We wish to upper bound the number of nodes we can generate in a given degree family j with a distribution following the Property 2.6. This problem can be expressed in terms of finding coefficients satisfying Equation 2.7. Here we may take coefficients $\|a\|_1 = 1$ and then generalize by taking $x = Na = [N_1, N_2, \cdots, N_m]^T$.

$$\max_a \sum_{k=1}^{m} k^j \frac{e^{-k}}{j!} a_k \tag{2.7}$$

We can see the maximum occurs when a_j has a maximal population. This means that, perhaps as expected, the way to achieve the maximum number of nodes with degree j is to maximize the number of input nodes with degree j. Since our function is nonincreasing this means this maximum occurs when $a_1 = a_2 = \cdots = a_j$ and $a_{j+1} = a_{j+2} = \cdots = a_m = 0$. This directly implies that we will get the most nodes of degree j when the following is true for $\|a\|_1 = 1$.

$$a_1 = a_2 = \cdots = a_j = \frac{1}{j} \tag{2.8}$$

Therefore the maximum number of nodes we should expect in a given degree class can be approximated as follows.

$$\frac{1}{j!j} \sum_{k=1}^{j} k^j e^{-k} \approx \frac{1}{j!j} \int_1^j x^j e^{-x} dx \tag{2.9}$$

$$\approx \frac{1}{j!j} \gamma(j+1, j) \tag{2.10}$$

Equation 2.9 gives us both the exact upper bound and continuous approximation. Equation 2.10 can be used as a quick approximation of this value in terms of the incomplete gamma function from 0 to j. This gives a far tighter bound than is provided by Equation 2.5 when we have a non-increasing degree distribution. It should be noted that one may improve upon the accuracy of these bounds for even more restrictive families of distributions

by including a lower bound as well as a tighter upper bound on the size of each degree family.

We can glean useful information from these bounds. For instance, if one desires an output distribution where more than a tenth of the nodes have degree five, there are no non-increasing inputs for which we should expect that property in output. In terms of the inverse matrix $\mathbf{P}^{-1}$, inputting such a vector will yield negative family sizes in some indices. This is incredibly limiting since this is independent of node number.

2.2.4 Results

We wish to determine how well $\mathbf{P}$ models the output of the Chung-Lu algorithm for a given input distribution. In Figure 2.3 we compare the naïve output distribution to the outputs of both Chung-Lu generation and our model taking that distribution as input. For this simple example we find that our model predicts the output node degree frequency remarkably well.

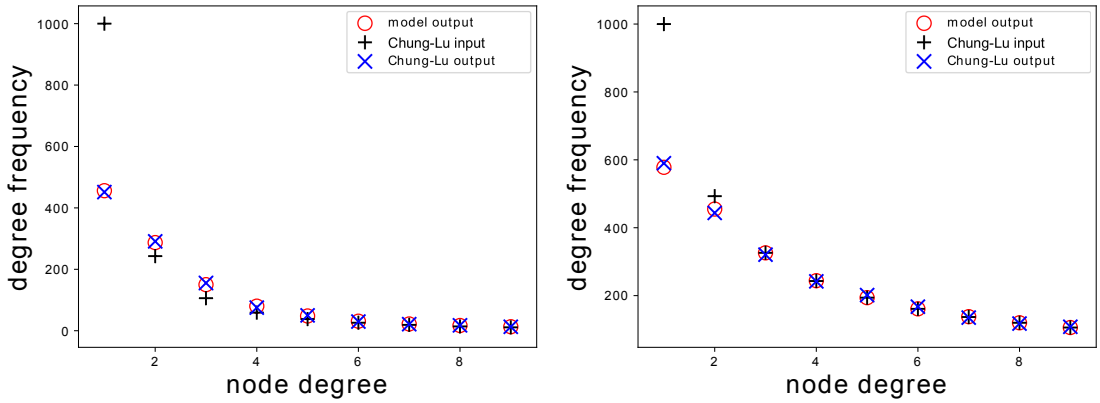


Figure 2.3: Model distribution versus Chung-Lu outputs. We consider the degree classes between one and nine for two different power law distributions. On the left is a power-law distribution with exponent $\beta = 1.0$ and on the right is a power-law distribution with exponent $\beta = 2.0$. In the top two plots, black crosses represent the naïve input $1000 \times k^{-\beta}$, red circles represent the distribution our model estimates will be the output of Chung-Lu generation, and blue x's represent the average distribution for 20 instances of Chung-Lu graphs given the black crosses as input. We can see that the Chung-Lu generated graphs match our model output remarkably closely.

Additionally we aim to determine how much proportional L1 accuracy is gained by using the vector $x = \mathbf{P}^{-1}y$ as opposed to y itself as an input to Chung-Lu. Specifically, we

consider generating a set of graphs $\{y_i\}$ using the Chung-Lu algorithm with the naïve inputs $\{y_i\}$, and the shifted inputs $\{x_i = \mathbf{P}^{-1}y_i\}$. We plot the proportional L1 errors $\frac{\|y_i-\tilde{y}_i\|_1}{\|y_i\|_1}$ and $\frac{\|y_i-\tilde{x}_i\|_1}{\|y_i\|_1}$ in Figure 2.4 where $\tilde{y}_i$ and $\tilde{x}_i$ are the output distributions of Chung-Lu for the naïve input and shifted input respectively. we choose our set $\{y_i\}$ such that these are guaranteed to be invertible distributions in the sense that $x \in \mathbb{R}^{+m}$. For this we use the variable precision toolbox in MATLAB with the digits of precision set to 100. The results of this can be seen in Figure 2.4. We find that our shifted input drastically decreases the proportional L1 error between the output of Chung-Lu and the desired output.

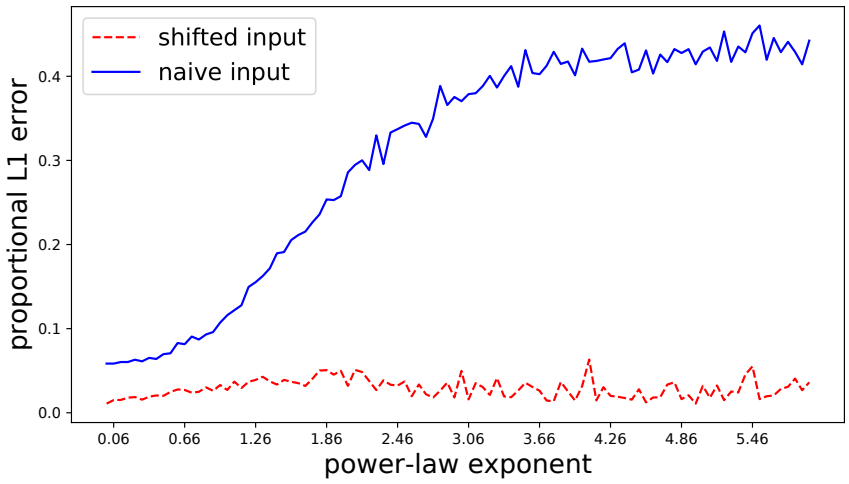


Figure 2.4: Error of naïve Chung-Lu input versus shifted Chung-Lu input. We consider 100 input distributions y_i such that $\mathbf{P}^{-1}y_i = x_i$ where the distribution x_i is the power-law distribution $1000 \times k^{-\frac{6i}{100}}$ with k ranging between 1 and 40. For each of the 100 inputs, 30 graphs were generated and their degree distributions were averaged using the input y_i for Chung-Lu. The proportional L1 error between this output and the desired output y_i is shown as the solid blue line. Additionally 30 graphs were generated and their degree distributions were averaged using the input x_i for Chung-Lu. The proportional L1 error between this output and the desired output y_i is shown as the dashed red line. We can see that the "shifted" input we get using our model drastically reduces error for the sample.

2.3 Correcting Chung-Lu

2.3.1 Introduction to correcting Chung-Lu

As discussed in the previous portion of this chapter, there are sometimes issues with the solution vectors given by "shifted Chung-Lu". These vectors do not have a useful interpretation with respect to the Chung-Lu generation algorithm, and the method is greatly limited because of this. The following section aims to remedy these issues while utilizing the same matrix model as an important building block.

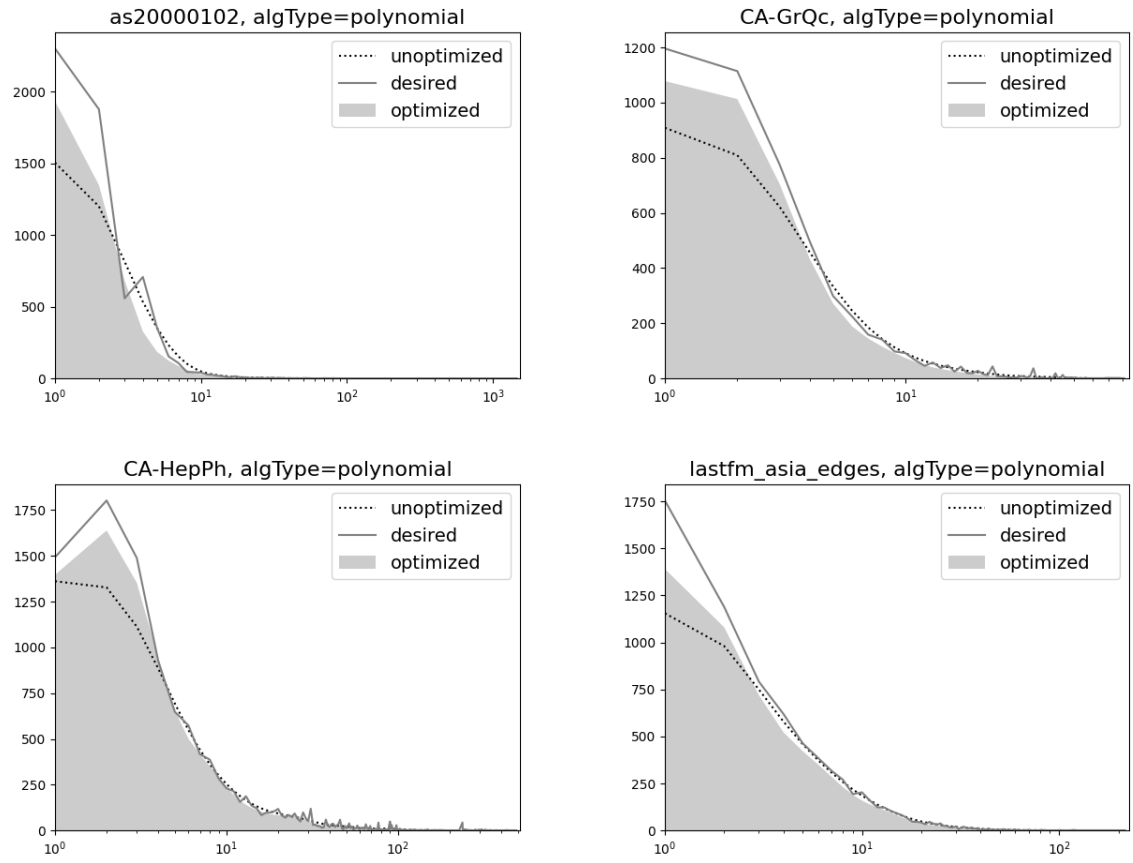


Figure 2.5: Visualization of degree sequences. Comparisons of degree sequences for the as20, GrQc, HepTh, and lastfm graphs. The dotted lines denote the predicted output using standard Chung-Lu weights, the solid grey region denotes the output sequence using optimized Chung-Lu weights, and the solid line denotes the desired output sequence. Each optimization here was performed using our polynomial update method.

We note that the matrix model can be easily generalized. By choosing a set of arbitrary positive weights $w = \{w_1, w_2, \cdots, w_d\}$, instead of simply the nodal degrees, one obtains a matrix $\mathbf{P}(w)$ where the means of each Poisson distribution correspond to the given weights. This produces a new stochastic block model.

2.3.2 Our contribution

This section focuses on determining, for a given number of weight parameters d, the set of weights such that the error between the desired output and actual output of Chung-Lu graph generation will be minimized. We develop and optimize several novel methods to minimize this error. Visualized in Figure 2.5 for several graphs in the Stanford Large Network Dataset Collection are their degree sequences, the unoptimized output from Chung-Lu generation using these degree sequences, and the generation output after applying one of our methods. We will discuss our varying methods in the following section and further analyze their results in the following sections.

2.3.3 Methods

As we note in our prior work [28], there are numerous sequences that can not be reliably generated using naïve Chung-Lu weights. To remedy this short coming, there are two algorithm parameters which may be adjusted to alter the output. One parameter is the input sequence. This is the specific parameter studied in prior work. The other parameter is the set of weights $w = \{w_1, w_2, \cdots, w_d\}$. Conceptually, both methods are trying to approximate a distribution as a linear combination of Poisson distributions. In the former method, the Poisson distributions have means equal to the target degree classes, and the approximation is improved by altering the coefficients applied to each distribution. Alternatively, changing the weights equates to changing these means, effectively moving the Poisson distributions along the x-axis.

We present two methods incorporating weight alteration. Our first method relies on several greedy updates, where weights are chosen such that $\|\mathbf{P}(w)x - y\|$ is minimized at each step. The latter method uses maximum likelihood estimation [29] to solve for weights.

Before discussing either method, let us first formalize goals and definitions. Take $\mathbf{P}$(w) to be the square matrix given by weights $w = \{w_1, \cdots, w_d\}$ and removing both the first

row and everything beyond row $d+1$ in Equation 2.2. The first row is removed because it corresponds to the number of zero-degree nodes. These nodes may be ignored after generation, so removing the first row of $\mathbf{P}(\mathrm{w})$ mathematically represents this. We call our input degree sequence vector $x = [x_1, \cdots, x_d]$ and our desired output degree sequence vector $y = [y_1, \cdots, y_d]$. Additionally, call the output of the Chung-Lu algorithm with weight set w and input vector x, $CL(w, x)$. Then, our goal is to find a combination w, x such that $\|CL(w, x) - y\|_1$ is minimized. The 1-norm is specifically considered because it can be directly interpreted as the number of nodes with incorrect degrees. A log_2-binned version of this error will additionally be considered later.

2.3.3.1 Greedy updates

We first discuss the greedy update method. This is based off of a simple approximation and update loop. The basic idea is as follows. Given an input degree sequence x, determine the first k derivatives of each column of $\mathbf{P}(\mathrm{w})$ with respect to their means and use these derivatives to approximate $\|\mathbf{P}(w+\epsilon)x - y\|$ for small perturbations in the elements of the mean-set $w + \epsilon$. Then, update the means in the optimal direction according to some minimization algorithm and repeat this process for some number of iterations.

Two update objectives are discussed in this section, which we call linear updates and polynomial updates. These objectives only differ in the number of derivatives considered. Linear updates approximate error based on the first derivative of each column in $\mathbf{P}(\mathrm{w})$. Alternatively, polynomial updates use an arbitrary number of k derivatives and the Taylor series to approximate error. As is shown later, both of these methods reduce the per-node degree error significantly; however, they require different numbers of iterations. All instances of the polynomial-update variant use $k = 2$ in this section. In Algorithm 2, we show a general template of the greedy method. The main difference in each of these methods comes from how our objective changes the opt_E$(\cdot)$ function. The objectives are discussed in more detail in the following subsections.

Algorithm 2 is initialized with an input degree sequence vector x, a desired output degree sequence vector y, a set of initial weights $\{w_1, \cdots, w_d\}$, a maximum update step-size δ, and an iteration number t. For this section, initial degree sequences are taken to be $x = cy$ for some positive constant $c \in \mathbb{R}^+$. Additionally, initial weights are taken to be $\{w_1 = 1, \cdots, w_d = d\}$. The iteration number and step size will vary depending on

Algorithm 2 Polynomial-Update (x,y,$\{w_1, \cdots, w_d\}$, δ,t)

1: $\mathbf{P} \leftarrow$ fill_P($\{w_1, \cdots, w_d\}$)
2: **for** iters $\in [1 \ldots \mathrm{t}]$ **do**
3: $\quad \mathbf{U} = \{\mathbf{U}_1, \cdots, \mathbf{U}_k\} \leftarrow compute\_U\_set(\{w_1, \cdots, w_d\})$
4: $\quad \mathbf{E} \leftarrow$ opt_E($\mathbf{P}$, $\mathbf{U}$, x, y, δ)
5: $\quad \mathbf{P} \leftarrow$ fill_P($\{w_1 + E_{11}, \cdots, w_d + E_{dd}\}$)
6: **return** $\{w_1, \cdots, w_d\}$

desired accuracy and whether linear, or polynomial updates are being used. The algorithm proceeds as follows. $\mathbf{P}$(w) is initialized with the input weights. Then, within the loop, a set of matrices $\mathbf{U} = \{\mathbf{U}_1, \cdots, \mathbf{U}_k\}$ is computed within the compute_U_set($\cdot$) function. Each matrix $\mathbf{U}_i$ corresponds to the i^{th} derivative of each column. These matrices are then used in the `opt_E(·)` function to determine how much each mean in w should change. For the purposes of this thesis, opt_E($\cdot$) uses the sequential least squares minimization [30] implementation from scipy.optimize.minimize($\cdot$) in Python. Then new weights are computed and $\mathbf{P}$(w) is updated.

2.3.3.2 Linear updates

Linear updates are the simpler of the two greedy update methods. In the linear update method, $k = 1$ and only a single $\mathbf{U}$ matrix is computed in compute_U_set($\cdot$). This matrix has the same form given in Equation 2.11 and the columns take the form of the derivatives of the columns in $\mathbf{P}$(w) as given in Equation 2.12 with respect to their means. In Equation 2.11, μ_j corresponds to the mean of the Poisson distribution.

$$\mathbf{U} = \begin{bmatrix} | & | & & | \\ \frac{\partial}{\partial \mu_1} poiss(\mu_1, x) & \frac{\partial}{\partial \mu_2} poiss(\mu_2, x) & \cdots & \frac{\partial}{\partial \mu_m} poiss(\mu_m, x) \\ | & | & & | \end{bmatrix} \tag{2.11}$$

$$\frac{\partial}{\partial \mu_i} poiss(\mu_i, x) = \frac{(x - \mu_i) e^{-\mu_i} \mu_i^{x-1}}{x!} \tag{2.12}$$

The linear update objective function used in opt_E($\cdot$) takes the form of minimizing $\gamma = \|(\mathbf{P}(w) + \mathbf{UE})x - y\|_2$ with respect to the diagonal matrix $\mathbf{E}$, where each entry is bounded by δ, $|E_{jj}| \leq \delta$. Unfortunately, linear approximations lack significant accuracy, and as such, the step size δ needs to be rather small to maintain stability within each optimization

step `opt_E(·)`. This ultimately leads to a method which requires many updates. This can be prohibitive for graphs with high maximum degree, since the dimensionality of our optimization problem depends on this.

2.3.3.3 Polynomial updates

The polynomial update method is very similar to the linear update method. In this method, higher order derivatives are considered in the Taylor series. This higher order error approximation is then used to predict degree sequence errors. The Taylor series approximation of the Poisson distribution is given by Equation 2.13.

$$poiss(z,x) = \frac{e^{-\mu}\mu^x}{x!} + \sum_{j=1}^{\infty}\left(\frac{\partial^j}{\partial\mu^j}poiss(\mu,z)\right)(z-\mu)^j \tag{2.13}$$

For a given number of derivatives k, a truncated series is used to make approximations. Note that the term on the left of the sum is an entry of the matrix $\mathbf{P}(\mathrm{w})$. Additionally, the right hand sum consists of two components, the j^{th} derivative, and a difference term. This allows us to rewrite this expression in terms of matrices as in Equation 2.14.

$$\mathbf{P}(w') \approx \mathbf{P}(w) + \sum_{j=1}^{k}\mathbf{U}_j\mathbf{E}_j \tag{2.14}$$

In Equation 2.14, $\mathbf{U}_j$ is the matrix corresponding to the j^{th} derivative of each column, similar to equation 2.11. $\mathbf{E}_j$ is a diagonal matrix with entries $E_j(a,a) = e_a^j$, corresponding to the step size in each dimension. In the polynomial update function, the error to be minimized is of the form $\gamma = \|(\mathbf{P}(w) + \sum_{j=1}^{k}\mathbf{U}_j\mathbf{E}_j)x - y\|_2$. Because of the increased accuracy of the polynomial method, a larger bound δ may be used for the step size. While we do not present bounds for this here, the size of δ can be chosen to be larger for larger instances of the number of derivatives k.

2.3.3.4 Maximum likelihood estimation

Maximum likelihood estimation (MLE) based clustering is a popular statistical method for determining probabilistic clusters for a data set [29]. Given a pre-defined type of statistical distribution (e.g. normal, binomial, Poisson, etc.) and a number of distributions m, MLE

clustering determines the parameters and coefficients for those distributions such that their mixture distribution has the highest likelihood of generating the data set. While MLE is most commonly used for clustering data, we instead use it here for function approximation. Consider the desired degree sequence y as a realization of a mixture distribution and the underlying statistical distributions as Poisson distributions. Then, the coefficients and means which are output as a mixture model from MLE may be interpreted as the input vector x and the μ values in $\mathbf{P}$(w) , respectively.

Algorithm 3 MLE-Update (m,y,d,iters)

1: x $\leftarrow [\frac{1}{m}, \cdots, \frac{1}{m}]$
2: $p_y \leftarrow \frac{y}{\|y\|_1}$
3: $\mu \leftarrow [\frac{d}{m}, \frac{2d}{m}, \cdots, d]$
4: (x,μ) $\leftarrow$ poiss_EM(x,μ,p_y,iters)
5: x $\leftarrow \|y\|_1 x$
6: **return** (x,μ)

Our MLE based method proceeds as follows, and is demonstrated in pseudocode in Algorithm 3. Begin by considering a desired output sequence y, a number of means m, and an interval $[0, d]$. Initialize a vector $x = [\frac{1}{m}, \cdots, \frac{1}{m}]$ and a vector of means $\mu = [\mu_1 = \frac{d}{m}, \mu_2 = \frac{2d}{m}, \cdots, \mu_m = d]$. Note that these means may be initialized randomly within the interval $[0, d]$, if desired. Then, normalize y to obtain a probability distribution $p_y = \frac{y}{\|y\|_1}$, from which points are sampled for maximum likelihood estimation. Maximum likelihood estimation is then run on these inputs, updating the entries of x and μ at each iteration. Once this has concluded, x is scaled by $\|y\|_1$ and each entry is rounded to the nearest natural number. This ensures that x now corresponds to the number of nodes instead of a proportion of all nodes.

As discussed earlier, there are two parameters which may be tuned when improving Chung-Lu graph generation. While our earlier work focused on changing the input sequence, and both the linear and polynomial methods focus on changing the means of Poisson distributions, Algorithm 3 simultaneously solves for both. Additionally, expectation maximization has a tune-able dimensionality. This means that one may take small samples from p_y, and consider fewer Poisson distributions to improve compute time. This is not an option that is readily available in the case of greedy linear and polynomial updates.

2.3.4 Results

In Figure 2.6, the three methods discussed in the previous section are compared against naïve Chung-Lu generation on a set of degree sequences from the Stanford Large Network Dataset Collection. Graph generation is performed using the `expected_degree_graph(·)` function from the `NetworkX` [31] package in `Python`.

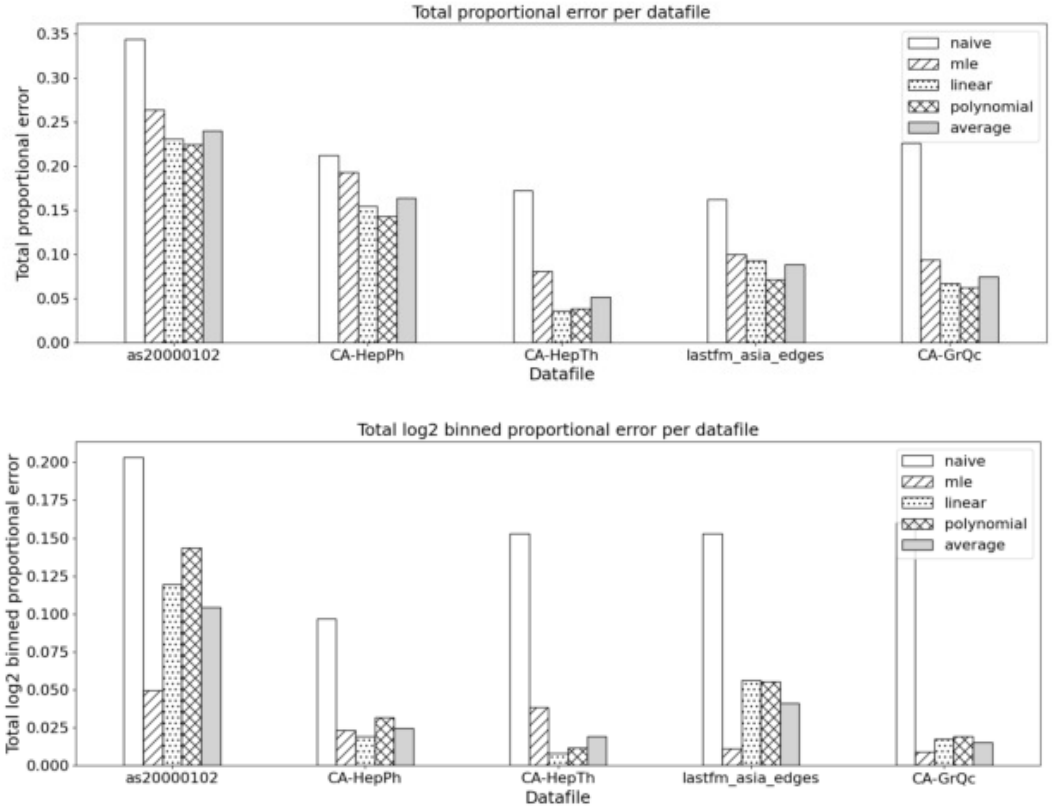


Figure 2.6: Proportional errors of degree sequences. Degree error plots for all methods on a number of graphs. Both the proportional error (top), and log_2-binned proportional error (bottom) metrics are as described in the Results section. As is seen, every method drastically reduces the proportional L1 error of the degree sequence when compared with naïve Chung Lu. However, different methods perform better on differing degree sequences.

As can be seen, each method outperforms naïve Chung-Lu by a considerable margin. However, our different methods perform better on different degree sequences. The exact reason for this requires further analysis. Figure 2.6 considers two different proportional error functions. The first one is L1 proportional error which is computed as the ratio

$\|CL(w,x) - y\|_1 / \|y\|_1$. This can be directly interpreted as the proportion of nodes which have the correct degree. Additionally, one can interpret this error function as a normalized version of the total variation distance. The log_2-binned proportional error is also considered. In this case the sequences $CL(w,x)$ and y are partitioned into $b = \lceil log_2(d) \rceil$ bins, forming the sequences $\beta(CL(w,x))$ and $\beta(y)$, both of which are in $\mathbb{R}^b$. The entries of $\beta(CL(w,x))$ are $\beta(CL(w,x))_i = \sum_{j=2^{(i-1)}}^{2^{(i-1)}+2^i} CL(w,x)_j$, and the entries of $\beta(y)$ follow similarly. The proportional binned error is then computed as $\|\beta(CL(w,x)) - \beta(y)\|_1 / \|\beta(y)\|_1$. The reason for defining this error function is that there are many applications where the exact degrees are less important than simply having the correct number of "low-degree", or "high-degree" nodes. For this purpose, the log_2-binned proportional error provides a quantitative understanding of how many nodes are being generated for different "sections" of the sequence.

As is seen in Figure 2.7, the polynomial update method outperforms the other optimization methods in proportional error. Additionally, the MLE optimization method outperforms the others for log_2-binned proportional error. Conceptually, this implies that the polynomial update method may be the best at matching the degrees of nodes exactly, while the MLE method is superior for approximate reproduction of sequences.

2.4 Discussion

2.4.1 Parameters

When choosing parameters for Algorithm 2, a reader may be rightfully curious as to what constitutes a "good" choice. In Figure 2.8, a parameter search over several choices of iteration number t and constant c, such that $x = cy$ are shown for two example graphs from the Stanford Large Network Dataset Collection. As is seen, the error reaches similar levels for both the polynomial and linear update methods for different parameters. We note that $1.05 < c < 1.15$ appears to work best for both graphs. While not shown, this behavior is also seen a across many other degree sequences. Furthermore, the number of iterations required to achieve a similar error reduction with polynomial updates versus linear updates is seen to be considerably smaller. In fact, for these two graphs, a similar error reduction is seen with an order of magnitude fewer update steps.

There is significant work to be done deciding parameters. While Figure 2.8 suggests

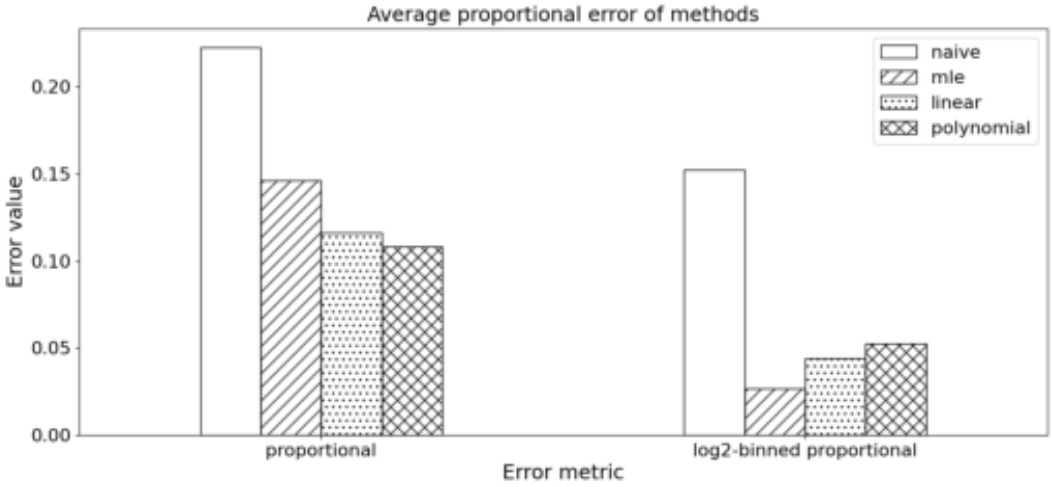


Figure 2.7: Average proportional errors of degree sequences. The proportional (left) and log2-binned proportional (right) errors are compared over all test graphs for each optimization method as well as naïve Chung-Lu. Both the proportional error, and log_2-binned proportional error metrics are as described in the Results section. As is seen, on average the polynomial update method results in the more significant reduction of proportional error, however the MLE method results in the largest reduction in log_2-binned proportional error.

some best practices, it is far from definitive. Furthermore, the choice of step-size δ is currently somewhat arbitrary. In this section, it is taken to be $0.05 \leq \delta \leq 0.2$ for linear updates, and $0.2 \leq \delta \leq 0.5$ for polynomial updates. Different step sizes drastically alter the stability and number of requisite iterations of the method. This requires further experimental and theoretical results for varying degree sequences.

2.4.2 Timing considerations

The methods presented in this thesis require varying times to run. The linear update method uses a miniscule step size, and as such requires many iterations to terminate. This is a significant concern when the maximum degree of the desired output is large. This is because the maximum degree controls the dimensionality of the optimization step, which must be performed at every iteration. To this end, the polynomial update method can iterate with a larger step size, requiring less iterations. However, in the case of a significantly large maximum degree, the optimization step may still not be practical. The MLE-method does not

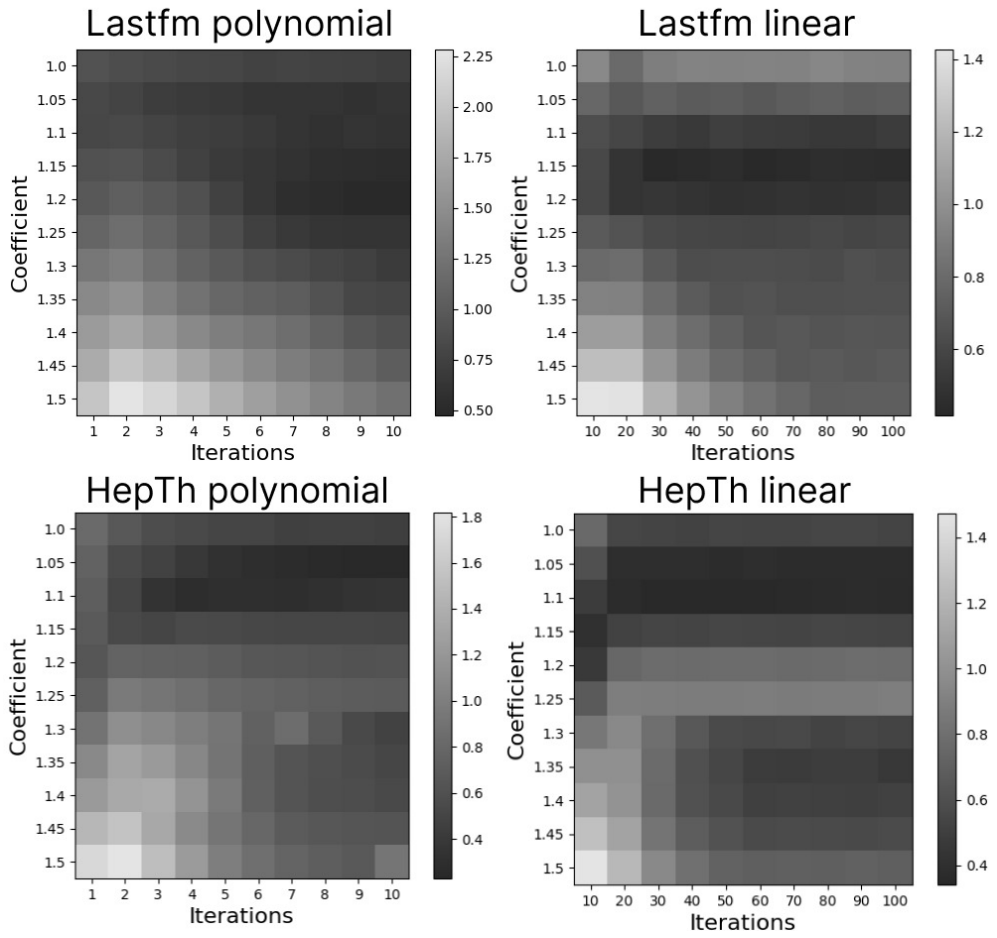


Figure 2.8: A parameter search of sequence error. We vary the coefficient $c \in \mathbb{R}^+$ for $x = cy$, and the number of iterations for both the polynomial and linear update methods respectively. The polynomial update method in this case has $k = 2$. The colors indicate the proportional L1 error $\|CL(w, x) - y\|_1 / \|CL(y) - y\|_1$. As can be seen for the two sample graphs, the polynomial update method converges to a smaller proportional L1 error than the linear method does in the same number of iterations.

suffer from these same drawbacks, because the sample number and number of distributions may be tuned. This means the MLE method should not perform slower on larger degree sequences, given constant sample and distribution numbers.

In the case of the greedy update methods, a simple change can be made which drastically speeds up compute time. This is the method of truncation. Note that, for most real world degree sequences the vast majority of the weight lies in the lowest degrees of the graph. Because of this, one may ignore a portion of the sequence when using either greedy update

method. This drastically reduces compute time, but may introduce additional error. In our limited testing, removing the final 1% of the sequence by node count greatly improves run times and minimally affects error. Despite this, the best practice for truncation is an open problem.

2.5 Extensions

Ultimately, the goal of improving graph null models is to make comparisons with observed datasets. For null models such as Chung-Lu graph generation, which are based on the degree distribution, these comparisons are useful for clustering tasks. In the case of modularity maximization [14] clusters are determined which maximize the Modularity heuristic 2.15.

$$Q = \frac{1}{2m} \sum_{u,v \in V} \left(A_{uv} - \frac{d_u d_v}{2m} \right) \delta(c_u, c_v) \tag{2.15}$$

In Equation 2.15, δ denotes the Kronecker delta, c_x denotes the cluster membership of node x, d_x denotes the degree of node x, and A_{xy} denotes the $x^t h, y^t h$ element of the adjacency matrix. The right hand term within the parenthesis in the equation 2.15 $d_u d_v / 2m$ is the same as the probability of two nodes with degree d_u and d_v being connected within the Chung-Lu model. We can therefore construct a more general modularity expression for any null model using per-edge probabilities as follows.

$$Q = \frac{1}{2m} \sum_{u,v \in V} \left(A_{uv} - p(u,v) \right) \delta(c_u, c_v) \tag{2.16}$$

In Equation 2.16, $p(u, v)$ denotes an arbitrary probability for the edge connecting nodes u, v. This has been discussed in Fosdick et al. [9], however they suggest an edge-skipping technique for computing these probabilities. This technique works as follows. First, a graph with the desired degree distribution is taken as an input and a copy of its adjacency matrix is stored. This graph can in general be taken as the one we want to cluster. Then double-edge swaps are performed. The exact number of double edge swaps required is unknown, but some number of double edge swaps are done. After these edge swaps, the adjacency matrix of the new graph is added to the prior adjacency matrix. These steps are repeated for some number of samples, and then the sum of adjacency matrices is divided by that number of samples to

obtain per-edge probabilities. In general this is a very slow technique, and anything which could achieve a similar result with grater speed could yield a more representative version of modularity maximization.

CHAPTER 3
SPECTRAL GRAPH COARSENING

3.1 Chapter overview

We investigate the problem of coarsening graphs while preserving the properties of their spectrum Laplacian spectrum. The first portion of this chapter is spent on analyzing the inverse problem for the unnormalized graph Laplacian. For this portion, we ask the question, *If a coarsened graph's spectrum approximates the spectrum of the original graph to within some threshold, may we approximately recreate the original graph from its coarsened counterpart with some bound on the per-edge weight difference?*. After this, we explore an algorithm for Laplacian consistent coarsening for the normalized Laplacian matrix on GPU.

3.2 Introduction to spectral coarsening

Graph coarsening has been a long standing field of study since Gabriel Kron's 1939 work [32] creating reduced order models of electrical networks. While this original work was focused mainly on coarsening for applications to electrical networks, it has spurred a great deal of study in several different disciplines, such as machine learning and scientific computing [33]. In machine learning and scientific computing, graph coarsening is often used as a pre-processing step for clustering or partitioning. An example of this is the METIS algorithm which partitions a coarsened graph before performing a series of refinement steps [34]. Recently, some work [35], [36] has put forward methods for coarsening graphs while preserving portions of the eigenspace relating to the graph Laplacian and normalized graph Laplacian [37]. There are many benefits to this sort of analysis for preserving broader functional behavior of a network. As opposed to preserving another metric, such as approximate

Portions of this chapter are to appear as: C. Brissette, A. Huang and G. M. Slota, "Spectrum consistent coarsening approximates edge weights.", *Soc. Ind. Appl. Math. J. Matrix Anal. Appl.*, vol. 44, no. 3, pp. 1032–1046, Sep. 2023

Portions of this chapter appear as: C. Brissette, A. Huang, and G. M. Slota, "Parallel coarsening of graph data with spectral guarantees," 2022, *arXiv:2204.11757*.

cut values, preserving a portion of the Laplacian eigenspace also preserves a portion of the behavior of the discrete heat and wave equations on the graph.

It is well known in the continuous case that the Laplacian operator can confer a significant amount of information about geometry [38]–[40]. While not all information can be retrieved [40], one may wonder what similar methods may reveal in the discrete case. Fortunately, a great deal of geometric information is conveyed through the spectrum of graphs as well [37], [41]–[44]. One may ideally wish for there to be a way to uniquely determine a graph and its automorphisms [45] by its spectrum. Unfortunately, a complete characterization of which graphs can be determined by their spectrum is an open problem, and most classes of graphs that are known to be classified by their spectrum are either small or rather simple in their structure. However, some geometric quantities can be found. For instance, we can "hear" the volume of a graph, defined to be the sum of it's degrees, simply by adding together each eigenvalue of the graph Laplacian. Additionally, we can "hear" an approximation of the optimal conductance cut in a graph by considering the first nontrivial eigenvalue [37].

The aim of this section is to formalize and prove a statement similar to the following. If the spectrum of a graph G is close to that of its coarsened representation G_c, then the edge weights of G can be closely determined from those of G_c. The utility of such a statement is perhaps best explored through the geometry of data. Assume there exists a set of k points $\{a_i\}_{i\in[1..k]}$, $a_i \in \mathbb{R}^n$. Form a graph from these points using a concave or convex weighting function $w_{ij} = f(\|a_i - a_j\|_2)$. Given the ability to approximate these weights within some bound given a coarsening G_c, it follows that the pairwise distances between nodes are also approximated within some bound given the same coarsening. For such a weighting scheme, this implies that coarsenings which closely preserve eigenvalues also closely preserve nodal embeddings in $\mathbb{R}^n$ to within some perturbation of a rigid transformation. This is discussed further in the final discussion section. For now, we begin towards this goal by defining the coarsening of a graph with respect to a nodal partition.

DEFINITION 1. *Consider a weighted graph $G = (V, W)$ and a partition of its nodes into k disjoint sets, $P = \{V_1, \cdots, V_k\}$, $V = V_1 \cup V_2 \cup \cdots \cup V_k$. The* ***coarsened graph of G with respect to P****, G_c, is the loopy weighted graph given by collapsing each of these partitions to a single node $\{\nu_1, \cdots, \nu_k\}$. The adjacency matrix elements are given by $W^c_{\nu_i\nu_j} = \sum_{u\in V_i}\sum_{v\in V_j} W_{uv}$. For brevity, we will often leave out the explicit partition P, and instead we refer to G_c simply as the coarsening of G.*

This is the interpretation provided in Loukas [35] and Jin et al. [36] and allows for coarsening can be expressed as a product of matrices $W_c = SWS^T$ for a coarsening matrix S. This definition retains the sum of weighted degrees within and between partitions, as well as the total sum of weighted degrees of the graph. One should note that the coarsened graph will have fewer eigenvalues than in the original graph. Comparing the spectra becomes difficult in this instance. For this reason, Loukas considers a truncated spectrum that is cut to the dimension of the coarsened graph. We instead follow Jin et al.'s later work, where the original and coarsened graphs are compared through a structure called the lift, which extends the spectrum of the coarsened graph to the correct dimension.

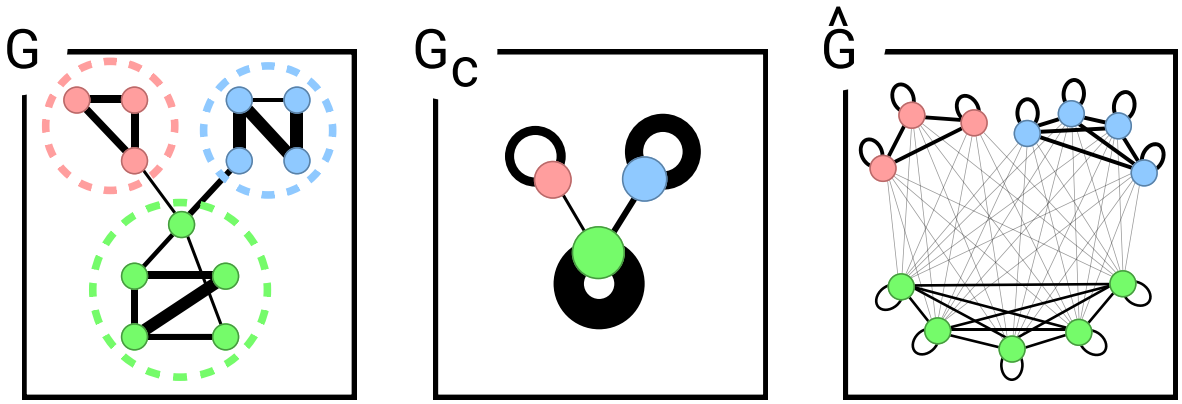


Figure 3.1: Coarsening and lifting. A visualization of the coarsening and lifting process. In the figure, the relative thickness of an edge positively correlates with the edge weight. Note how in the shift from G to G_c the graph gains self-loops. Additionally, after lifting the coarsened graph G_c to $\hat{G}$, the weights within and between partitions become evenly distributed.

Definition 2. *Consider a coarsening G_c of graph $G = (V, W)$ with respect to nodal partition $P = \{V_1, \cdots, V_k\}$. We call $\hat{G} = (\hat{V}, \hat{W})$ the lift of G with respect to P, where $|\hat{V}| = |V|$. The adjacency matrix elements are given by $\hat{W}_{uv} = W^c_{\nu_i \nu_j}/(|V_i||V_j|)$ where $u \in V_i$ and $v \in V_j$. For brevity, we will often assume a partition P with associated coarsening G_c and simply refer to $\hat{G}$ as the lift of G.*

As previously mentioned, the lift is useful because the sorted eigenvalues of the lift $\hat{G}$ align with those of the original graph G. Before continuing we define the graph Laplacian as $L = D - W$, and the normalized Laplacian as $\mathcal{L} = D^{-1/2} L D^{-1/2}$, where D is the diagonal

degree matrix of G, and W is the weighted adjacency matrix. We now define notions which will be useful when comparing the structure of the original graph G with that of the lift $\hat{G}$. This begins with the notion of σ-connectedness.

Definition 3. *Consider a graph $G = (V, W)$ and a nodal partition $P = \{V_1, \cdots, V_k\}$. The weighted adjacency of G can be written as $W = W(C) + W(R)$. Here $W(C)$ is a block-diagonal matrix of k disconnected weighted adjacencies corresponding to the k induced subgraphs of G given by the entries in P. The matrix $W(R)$ is the adjacency of a weighted k-partite graph on the same partitions. Then for $\|W(R)\|_1 \leq \frac{\sigma}{2}$, we call the graph σ-connected.*

$$L = \begin{bmatrix} L_{11} & L_{12} & \cdots & L_{1k} \\ L_{21} & L_{22} & \cdots & L_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ L_{k1} & L_{k2} & \cdots & L_{kk} \end{bmatrix} = \begin{bmatrix} C_1 & 0 & \cdots & 0 \\ 0 & C_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & C_k \end{bmatrix} + \begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1k} \\ R_{21} & R_{22} & \cdots & R_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ R_{k1} & R_{k2} & \cdots & R_{kk} \end{bmatrix} \tag{3.1}$$

We will also refer to Laplcian C associated with $W(C)$ as the core Laplacian, and the Laplcian R associated with $W(R)$ as the ambient Laplacian. The associated graphs for these matrices will be referred to as the core structure $\mathcal{C}$, and ambient structure $\mathcal{R}$ respectively. There are several noteworthy properties of the core and ambient structures of a graph G given the nodal partition P. First, the adjacency matrix W_c for a coarsening of G with respect to partition P is equal to $W(C)_c + W(R)_c$, where $W(C)_c$ and $W(R)_c$ are the coarsened adjacency matrices of the core and ambient structures with respect to the same partition. Additionally, the lift $\hat{W}$ is equal to $\hat{W}(C) + \hat{W}(R)$, where $\hat{W}(C)$ and $\hat{W}(R)$ are the lifted adjacency matrices of the core and ambient structures with respect to the partition P. Note that all graphs are σ-connected for some value σ. Because of this, all graphs can be broken into a core and ambient structure, where the ambient structure defines perturbations in the Laplacian of the core structure C. This is largely how this definition is used as the section progresses.

Definition 4. *We call a weighted graph $G = (V, W)$ δ-complete if all the weights in the graph are $W_{uv} = \frac{\delta}{N}$.*

The definition of a δ-complete graph generalizes the concept of the complete graph in the unweighted case to one in the weighted case. It is worth noting that δ-complete graphs have the same normalized Laplacian spectrum as complete graphs. Additionally, the

nontrivial eigenvalues of the combinatorial Laplacian for δ-complete graphs are all equal to δ. Graphs which are δ-complete form the natural building blocks of the lift of the core structure $\hat{\mathcal{C}}$. With this, all the requisite language is defined.

3.2.1 Notation

Graph $G = (V, W)$ is assumed to be weighted, and $|V| = N$, $|W| = M$. The variable ϵ will refer to a real number such that $\epsilon > 0$. Additionally we will be discussing many eigenvalues of different matrices. The ordered eigenvalues of the Laplacians of G and $\hat{G}$ will be denoted as λ_i and $\hat{\lambda}_i$ respectively. Similarly the ordered eigenvalues of the adjacency matrices W and $\hat{W}$ associated with G and $\hat{G}$ will be denoted ω_i and $\hat{\omega}_i$ respectively. Additionally we will concern ourselves with the normalized Laplacians $\mathcal{L}$ and $\hat{\mathcal{L}}$. The eigenvalues of these will be denoted η_i and $\hat{\eta}_i$ respectively. The eigenvalues of the core Laplacian C with k connected components will be denoted $\mu_i(k)$, where (k) denotes membership within the assumed partition $P = \{V_1, \cdots, V_k\}$. the associated eigenvalues of the lifted core Laplacian $\hat{C}$ will be denoted by $\hat{\mu}_i(k)$.

Weighted adjacency matrices will be denoted by W and $\hat{W}$ respectively. Individual adjacencies between nodes $u, v \in V$ will be denoted by W_{uv}, and $\hat{W}_{uv}$ will denote the adjacency between $u, v \in \hat{V}$. Additionally $M_{i:}$ and $M_{:i}$ represent the i^{th} and column respectively for an arbitrary matrix M. The degree of any node $u \in V$ will be denoted by d_u. Similarly, $\hat{d}_u$ will denote the same for $u \in \hat{V}$. These degrees show up in the diagonal degree matrices D and $\hat{D}$. Additionally, when considering eigenvalues of induced subgraphs with respect to some partition $P = \{V_1, \cdots, V_k\}$, they will be expressed as $\lambda_i(j)$ where $j \in [1..k]$ denotes set membership within an element of P. This notation extends to degrees, as well as all other associated eigenvalues. Finally $vol(H)$, for some subgraph H, denotes the sum of weighted degrees within the subgraph.

3.2.2 Spectrum consistent coarsening

We first present a method for spectrum consistent coarsening of a graph G with respect to the combinatorial Laplacian. This is in contrast to the work in Jin et al. [36] using normalized Laplacians; however, the proof method is incredibly similar. The idea behind this method is simple. Two nodes may be merged if their rows in the adjacency matrix are approximately

linearly dependent. This linear dependence is evaluated by computing the 1-norm of the difference between rows in the adjacency matrix, and merging the rows with the smallest 1-norm difference. This is then iterated to the users desired level of coarsening. The proof follows the same format as Proposition 4.2 in Jin et al.

THEOREM 5 (SPECTRUM CONSISTENT COARSENING). *For a graph G with all self-loops having the same weight, if it is coarsened by combining nodes $u, v \in V$, then $|\lambda_i - \hat{\lambda}_i| \leq \frac{3\epsilon}{2}$ if $\|W_{u:} - W_{v:}\|_1 \leq \epsilon$.*

Proof. The proof follows similarly to that in Jin et al. We consider entries of the lifted adjacency matrix.

$$\hat{W}_{ij} = \begin{cases} \frac{W_{uu}+W_{uv}+W_{vu}+W_{vv}}{4} & \text{if } i,j \in \{u,v\} \\ \frac{W_{uj}+W_{vj}}{2} & \text{if } i \in \{u,v\} \text{ and } j \notin \{u,v\} \\ \frac{W_{iu}+W_{iv}}{2} & \text{if } i \notin \{u,v\} \text{ and } j \in \{u,v\} \\ W_{ij} & else \end{cases}$$

As noted in the original citation, this then means that the degrees of the lifted nodes will be as follows.

$$\hat{d}_i = \begin{cases} \frac{d_u+d_v}{2} & \text{if } i \in \{u,v\} \\ d_i & else \end{cases}$$

There are now a Laplacian $L = D - W$ and a lifted Laplacian $\hat{L} = \hat{D} - \hat{W}$, and we wish to know the difference between these $E = L - \hat{L} = D - \hat{D} + \hat{W} - W$ as to apply Weyl's inequality.

$$\hat{W}_{ij} - W_{ij} = \begin{cases} \frac{W_{uu}+W_{uv}+W_{vu}+W_{vv}}{4} - W_{ij} & \text{if } i,j \in \{u,v\} \\ \frac{W_{uj}+W_{vj}}{2} - W_{ij} & \text{if } i \in \{u,v\} \text{ and } j \notin \{u,v\} \\ \frac{W_{iu}+W_{iv}}{2} - W_{ij} & \text{if } i \notin \{u,v\} \text{ and } j \in \{u,v\} \\ 0 & else \end{cases}$$

$$D_{ii} - \hat{D}_{ii} = \begin{cases} d_i - \frac{d_u+d_v}{2} & \text{if } i \in \{u,v\} \\ 0 & else \end{cases}$$

Because $\|W_{u:} - W_{v:}\|_1 \leq \epsilon$, it is also true that $|d_u - d_v| \leq \epsilon$ by the triangle inequality. Therefore, without loss of generality, $\frac{d_u+d_v}{2} \leq d_u + \frac{\epsilon}{2}$ meaning $|d_u - \frac{d_u+d_v}{2}| \leq \frac{\epsilon}{2}$. Therefore, to prove the lemma, only considerations for the difference in the adjacency matrices remain. For this, two cases need to be analyzed, including the case where $i \in \{u, v\}$ and the case where $i \notin \{u, v\}$. Without loss of generality, take $i = u$, then in the former case above the following is true.

$$\begin{aligned}
\|\hat{W}_{i:} - W_{i:}\|_1 &= \sum_{j \in V} |\hat{W}_{uj} - W_{uj}| \\
&= \left|\frac{W_{uu} + W_{uv} + W_{vu} + W_{vv} - 4W_{uu}}{4}\right| + \left|\frac{W_{uu} + W_{uv} + W_{vu} + W_{vv} - 4W_{uv}}{4}\right| \\
&+ \sum_{j \notin \{u,v\}} \left|\frac{W_{uj} + W_{vj} - 2W_{uj}}{2}\right| \\
&= \frac{1}{4}|W_{uv} + W_{vu} + W_{vv} - 3W_{uu}| + \frac{1}{4}|W_{uu} + W_{vv} - 2W_{uv}| + \frac{1}{2}\sum_{j \notin \{u,v\}} |W_{vj} - W_{uj}| \\
&\leq \frac{3}{4}|W_{uu} - W_{uv}| + \frac{3}{4}|W_{vv} - W_{uv}| + \frac{1}{4}|W_{uu} - W_{vv}| + \frac{1}{2}\sum_{j \notin \{u,v\}} |W_{vj} - W_{uj}| \\
&\leq |W_{uu} - W_{uv}| + |W_{vv} - W_{uv}| + \frac{1}{2}\sum_{j \notin \{u,v\}} |W_{vj} - W_{uj}| \\
&\leq \|W_{u:} - W_{v:}\|_1 \leq \epsilon
\end{aligned}$$

Now we prove a similar result when $i \notin \{u, v\}$.

$$\begin{aligned}
\|\hat{W}_{i:} - W_{i:}\|_1 &= \sum_{j \in V} |\hat{W}_{ij} - W_{ij}| \\
&= \frac{1}{2}|W_{iu} - W_{iv}| + \frac{1}{2}|W_{iv} - W_{iu}| \\
&= |W_{ui} - W_{vi}| \\
&\leq \|W_{u:} - W_{v:}\|_1 \leq \epsilon
\end{aligned}$$

Then $\|E\|_1 = \|\hat{L} - L\|_1 = \|\hat{D} - D + W - \hat{W}\|_1 \leq \frac{\epsilon}{2} + \epsilon = \frac{3\epsilon}{2}$. Our lemma then follows immediately from Weyl's inequality.

□

3.2.3 Discussion

We note that this bound can be repeated in succession m times, and if each successive coarsening has an L1 difference less than or equal to ϵ, then the spectral gap we obtain at the end between our graphs is less than or equal to $\frac{3m\epsilon}{2}$. We show an example of this coarsening performed on a σ-connected graph according to the criteria of theorem 5 in figure 3.2.

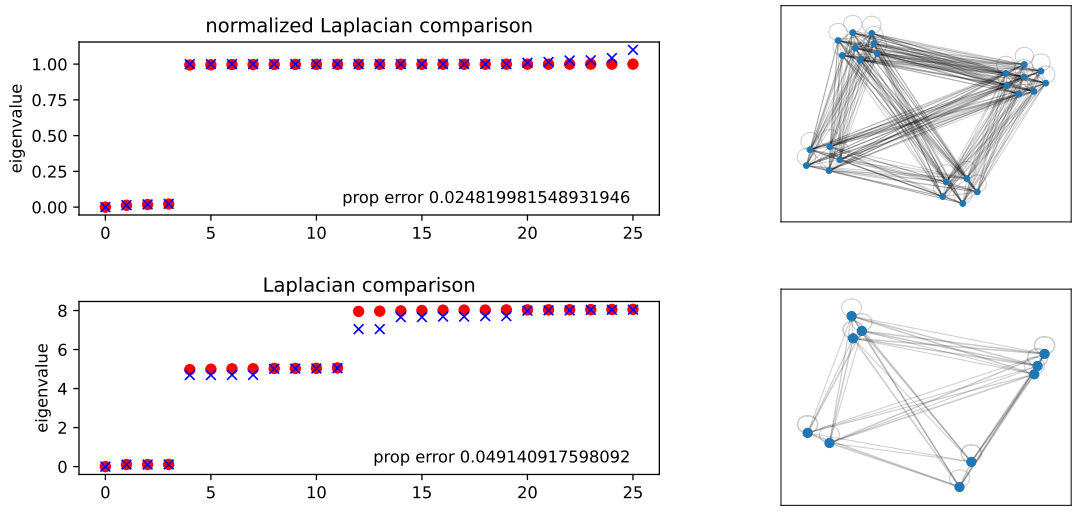


Figure 3.2: A coarsening example. An example of σ-connected graph is pictured in the top right of the figure and a greedily coarsened representation is shown beneath it. This graph was coarsened according to the criteria in Theorem 5, using $\epsilon \leq 0.1$ as a maximum threshold. The red-dots denote eigenvalues of the original graph, and the blue crosses denote the eigenvalues of the lift after coarsening. Both the spectrum of the normalized Laplacian and the spectrum of the combinatorial Laplacian are shown along with the proportional L1 error for each of them as $\|\Lambda - \hat{\Lambda}\|_1 / \|\Lambda\|_1$, where Λ and $\hat{\Lambda}$ are vectors containing the sorted eigenvalues of G and $\hat{G}$ respectively.

When coarsening with respect to the normalized Laplacian, the eigenvalues of the coarsened graph are all eigenvalues of the lifted graph. This makes comparisons between the lift and the original graph meaningful, because they bound the spectral behavior of the coarse graph. This is not true in the case of the combinatorial Laplacian. However, coarsening with respect to the combinatorial Laplacian achieves multiple objectives.

THEOREM 6. *Given a graph $G = (V, W)$, if two nodes $u, v \in V$ are such that $\|W_{u:} - W_{v:}\|_1 \leq \epsilon$, then $\|W_{u:}/d_u - W_{v:}/d_v\|_1 \leq 2\epsilon/max\{d_u, d_v\}$.*

Proof. First note that $\|W_{u:} - W_{v:}\|_1 \leq \epsilon$ implies $|d_u - d_v| \leq \epsilon$. Then without loss of generality the following is true.

$$\begin{aligned}\left\| \frac{W_{u:}}{d_u} - \frac{W_{v:}}{d_v} \right\|_1 &= \left\| \frac{d_v W_{u:} - d_u W_{v:}}{d_u d_v} \right\|_1 \\ &\leq \left\| \frac{W_{u:} - W_{v:}}{d_v} \right\|_1 + \frac{\epsilon}{d_v} \leq \frac{2\epsilon}{d_v}\end{aligned}$$

Because d_v was chosen without loss of generality, $\|W_{u:}/d_u - W_{v:}/d_v\|_1 \leq 2\epsilon/max\{d_u, d_v\}$ is true, proving the theorem.

□

As shown in Theorem 6, the bound ϵ given in combinatorial Laplacian coarsening enforces a related bound with regards to normalized Laplacian coarsening. That is, combinatorial Laplacian coarsening coarsens with respect to both objectives at once. This is not guaranteed with normalized Laplacian coarsening. Furthermore, as is discussed throughout this manuscript, relations between nodes in the fine graph are preserved when coasening with respect to the combinatorial Laplacian. This has potential applications to data mining and clustering tasks.

We now present a comparison between the normalized objective and algebraic distance. AMG methods are extensions of classical multigrid methods [46] in scientific computing, which rely on successive coarsenings of a linear operator to efficiently solve a linear system. The algebraic distances used in relaxation based AMG methods rely on sampling a set of test vectors [47] $\{x^i\}_{i\in[1..k]}$ and computing $\chi^i = L_{rw}x^i$, where L_{rw} is the random walk normalized Laplacian [37]. A distance metric between nodes is computed using the output of these vectors. One such metric is the following.

$$\alpha_{uv} = \underset{i\in[1..k]}{max} |\chi_u^i - \chi_v^i| \tag{3.2}$$

Intuitively, this quantifies the linear dependence between rows of the random walk Laplacian L_{rw}. In Jin et al. the linear dependence between rows of the random walk Laplacian is also considered. In the former case, linear dependence is probed by test vectors, whereas it is

quantified by the 1-norm difference in the latter case. Using the bound in Jin et al. one can also bound the distance metric α_{uv}. Assuming $\|x^i\|_2 = 1$ for all $i \in [1..k]$, then the following is true.

$$\left\| \frac{W_{u:}}{d_u} - \frac{W_{v:}}{d_v} \right\|_1 \leq \epsilon \tag{3.3}$$

$$\Rightarrow \left| \left(\frac{W_{u:}}{d_u} - \frac{W_{v:}}{d_v} \right) x^i \right| \leq \epsilon \|x^i\|_2 \leq \epsilon \tag{3.4}$$

$$\Rightarrow \alpha_{uv} \leq \epsilon \tag{3.5}$$

One may construct a similar algebraic distance using the combinatorial Laplacian. In this case, the same argument can be made using the bound given in Theorem 5. A very similar bounding argument may be made for another algebraic distance metric used in AMG methods.

$$\beta_{uv} = \sum_{i \in [1..k]} \left(\chi_u^k - \chi_v^k \right)^2 \tag{3.6}$$

If the bound from Jin et al. is assumed, one knows $\beta_{uv} \leq k\epsilon^2$ due to a similar argument as before. An identical argument can be made for an algebraic distance using the combinatorial Laplacian when the bound given in Theorem 5 is used. In this sense, the bound from Theorem 5 can be interpreted as a stronger criteria than those of relaxation based algebraic multigrid. To achieve the same spectral bound using an algebraic metric such as α_{uv} or β_{uv}, one requires $k = N$ linearly independent test vectors at minimum. This implies the work complexity of relaxation based AMG needs to be $O(MN)$ to achieve a similar spectral bound to Theorem 5. Alternatively, computing a sparse norm between adjacency vectors for every edge in the graph requires only $O(N \langle d^2 \rangle)$ where $\langle d^2 \rangle$ is the second moment of the graph's degree distribution [48]. While it may be possible to define a spectral bound given algebraic distances between nodes, to the authors' knowledge no such bound exists [33]. Figure 3.3 compares coarsening heuristics on three different graphs from the Koblenz Network Collection. In Figure 3.3 one can see that, despite not having the same guarantees as the criteria in Theorem 5, coarsening with respect to algebraic distance does perform well for spectral approximation. Both methods perform better than heavy weight matching, which

is a popular coarsening method attempting to maximize $\frac{w_i j}{d_i d_j}$ for each merge [33].

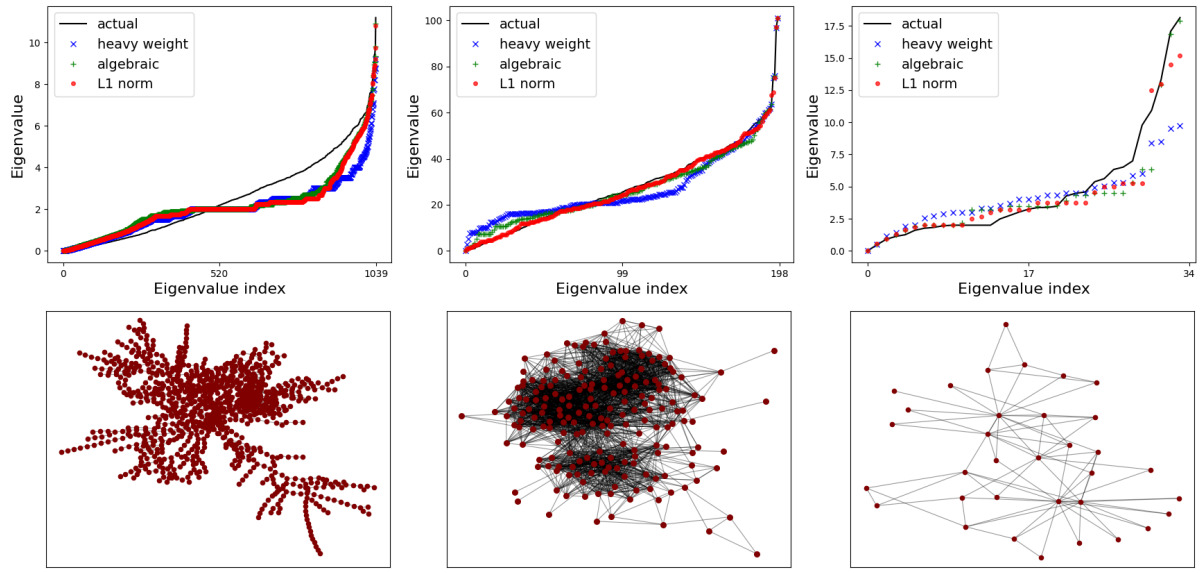


Figure 3.3: Coarsening method comparison. Three graphs were coarsened using three different heuristics to half-size and then the spectra of their lifts were compared with the original. From left to right, the graphs are Euroroad, Jazz Musicians, and the Zachary Karate Club. All of these were collected from the Koblenz Network Collection. The bottom row of the figure displays the original graphs. For heavy weight matching the edge (u, v) corresponding to the highest value of $\frac{W_{uv}}{d_u d_v}$ was contracted at each step. For algebraic distance, the edge corresponding to the minimum of Equation 3.2 was contracted. Finally for the L1 method, the edge minimizing the criteria in Theorem 5 was contracted. It should additionally be noted that 20 test vectors were used for computing algebraic distances.

3.2.4 Edge weight approximation for general graphs

We now proceed with the proof of the main result of the manuscript. The details behind the main result of this chapter are that every graph is σ-connected for some value σ and the Laplacian can be expressed as the sum of two independent graph Laplacians. These Laplacians are defined by the core Laplacian C and the ambient Laplacian R. Comparisons can then be made between the core structures of the original graph and the lift, viewing R and $\hat{R}$ as perturbation matrices. In this way the degrees of individual nodes may be

bounded according to the difference in spectra, and the parameter σ. This allows for the application of discrepancy bounds [37] to various subgraphs and, most importantly, pairs of nodes. These bounds are then used to bound weight differences accordingly. A visualization of the dependencies between results can be seen in Figure 3.4.

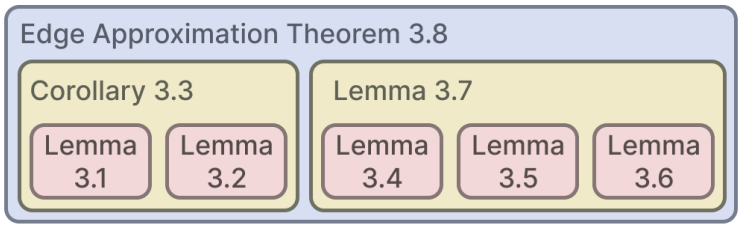


Figure 3.4: Result dependencies. The dependencies between results in this section are shown. The boxes are numbered with their respective results, and any boxes nested within them represent sub-results which are used to prove the larger corollary, lemma, or theorem.

LEMMA 7. *Given a σ-connected graph $G = (V, W)$ and its approximated lifted graph $\hat{G} = (\hat{V}, \hat{W})$, say $|\lambda_i - \hat{\lambda}_i| \leq \epsilon$ for all $i \in [1..N]$. Then $|\mu_i - \hat{\lambda}_i| \leq \epsilon + \sigma$ for all $i \in [1..N]$.*

Proof. Since G is σ-connected, its Laplacian structure can be written as the sum of two separate Laplacians, $L = C + R$. Note $\|R\|_2 \leq \|R\|_1 \leq \sigma$. Using this, we directly apply Weyl's inequality [49] to get $|\lambda_i - \mu_i| \leq \sigma$. Then, because $|\lambda_i - \hat{\lambda}_i| \leq \epsilon$, $|\mu_i - \hat{\lambda}_i| \leq \epsilon + \sigma$. □

In simplifying language, Lemma 7 states that the eigenvalues of the core Laplacian C closely approximate eigenvalues of the lift $\hat{G}$ when the eigenvalues of $\hat{G}$ closely approximate the eigenvalues of G.

LEMMA 8. *If $G = (V, W)$ is a σ-connected graph, then its lift $\hat{G} = (\hat{V}, \hat{W})$ is also σ-connected.*

Proof. Note that lifting preserves the sum of edge weights within partitions and the sum of edge weights between partitions. Therefore, if L is expressed in terms of its core and ambient structures, the lifts of both of these structures may be independently considered. For any node $u \in V_i$ in $\hat{\mathcal{R}}$, the degree $\hat{d_u} = \frac{1}{|V_i|} \sum_{v \in V_i} d_v$. The following relationship then holds true.

$$\max_{v \in V} |\hat{d}_v| \leq \max_{u \in V} |d_u|$$

This implies $\|\hat{R}\|_1 \leq \|R\|_1 \leq \sigma$ proving our lemma.

□

Lemma 8 in conjunction with Lemma 7 allows for direct comparison between the spectra of the core Laplacians C and $\hat{C}$.

COROLLARY 9. *Given a σ-connected graph $G = (V, W)$ and its approximated lifted graph $\hat{G} = (\hat{V}, \hat{W})$, respectively, say $|\lambda_i - \hat{\lambda}_i| \leq \epsilon$ for all $i \in [1..N]$. Then $|\mu_i - \hat{\mu}_i| \leq \epsilon + 2\sigma$.*

Proof. This follows immediately from the fact that $\hat{G}$ is σ-connected from Lemma 8, and then applying Lemma 7.

□

Corollary 9 states that the core-structure of a graph and its lift have similar eigenvalues when G and $\hat{G}$ have similar eigenvalues. Each independent core sub-Laplacian $\hat{C}(i)$ of our lift $\hat{G}$ is δ_i-complete, where δ_i is the average degree within $\mathcal{C}(i)$. This implies $\hat{\mu}_j(i) = \delta_i$ for all nontrivial eigenvalues. Therefore, full information of the degrees and spectra of every $\hat{\mathcal{C}}(i)$ are known. In conjunction with Corollary 9, this will allow for comparison between the degrees of partitions of the core structures $\mathcal{C}(i)$ and $\hat{\mathcal{C}}(i)$.

LEMMA 10. *If all the nontrivial eigenvalues of the Laplacian L of a connected graph $G = (V, W)$ lie within the bounds $\delta - \epsilon \leq \lambda_i \leq \delta + \epsilon$, with $\delta = \frac{Vol(G)}{N}$, then $|d_i - \delta| \leq 4\epsilon$ for all $i \in [1..N]$. Here $d_i = \sum_{j \in [1..N]} W_{ij}$ is the degree of the node $i \in V$.*

Proof. Consider the vector $e_{ij} = \frac{1}{\sqrt{2}}(e_i - e_j)$, where e_i, e_j are the unit vectors with value zero everywhere except for a one in the i^{th} and j^{th} element, respectively. Note that $e_{ij}^T \perp \mathbf{1}$, meaning $\|Le_{ij}\|_2$ cannot be arbitrarily small. Instead, it is bounded below and above by $\lambda_2 \leq \|Le_{ij}\|_2 \leq \lambda_N$. By assumption, $\delta - \epsilon \leq \lambda_i \leq \delta + \epsilon$ for all nontrivial eigenvalues. This means $\delta - \epsilon \leq e_{ij}^T L e_{ij} \leq \delta + \epsilon$ must be true due to e_{ij} having unit length $\|e_{ij}\|_2 = 1$. By writing out $e_{ij}^T L e_{ij}$ explicitly, one gets that $2(\delta - \epsilon) \leq (d_i + d_j + 2W_{ij}) \leq 2(\delta + \epsilon)$. From this, the following must be true.

$$2N(N-1)(\delta-\epsilon) \leq \sum_{i=1}^{N}\sum_{j=1,j\neq i}^{N} d_i + d_j + 2W_{ij} \leq 2N(N-1)(\delta+\epsilon)$$
$$2N(N-1)(\delta-\epsilon) \leq \sum_{i=1}^{N}(N-1)d_i + (Vol(G)-d_i) + 2d_i \leq 2N(N-1)(\delta+\epsilon)$$
$$2N(N-1)(\delta-\epsilon) \leq 2NVol(G) \leq 2N(N-1)(\delta+\epsilon)$$
$$\Rightarrow \frac{1-N}{N}\epsilon \leq \delta + \frac{1-N}{N}\delta \leq \frac{N-1}{N}\epsilon$$
$$\Rightarrow (1-N)\epsilon \leq \delta \leq (N-1)\epsilon$$

The inequality $\delta \leq (N-1)\epsilon$ can now be used to prove our lemma. The proof follows similarly to the previous inequalities, however now the outer sum is removed.

$$2(N-1)(\delta-\epsilon) \leq \sum_{j=1,j\neq i}^{N} d_i + d_j + 2W_{ij} \leq 2(N-1)(\delta+\epsilon)$$
$$2(N-1)(\delta-\epsilon) \leq (N-1)d_i + (Vol(G)-d_i) + 2d_i \leq 2(N-1)(\delta+\epsilon)$$
$$2(N-1)(\delta-\epsilon) \leq Nd_i + Vol(G) \leq 2(N-1)(\delta+\epsilon)$$
$$\Rightarrow 2\frac{(N-1)}{N}(\delta-\epsilon) \leq d_i + \delta \leq 2\frac{(N-1)}{N}(\delta+\epsilon)$$
$$\Rightarrow \frac{1-N}{N}\epsilon \leq d_i - \delta + \frac{2\delta}{N} \leq 2\frac{N-1}{N}\epsilon$$
$$\Rightarrow -2\epsilon - \frac{2(\delta-\epsilon)}{N} \leq d_i - \delta \leq 2\epsilon - \frac{2(\epsilon+\delta)}{N}$$
$$\Rightarrow -2\epsilon - 2\epsilon \leq d_i - \delta \leq 2\epsilon$$
$$\Rightarrow |d_i - \delta| \leq 4\epsilon$$

The second to last line comes as an immediate consequence of the previous inequality $\delta \leq (N-1)\epsilon$, and proves our lemma.

□

Lemma 10 allows for statements to be made about the degrees of nodes in G based on the average degrees of partitions. This completes one of two major building blocks for the final edge approximation theorem. Before proving the next lemma we state a weighted

version of Theorem 5.1 in Chung and Graham [37], noting that the original proof provided does not change in the case of weights.

LEMMA 11 (CHUNG. 5.1). *Suppose X, Y are two subsets of the vertex set V of a graph G. Then,*

$$\left| \sum_{x \in X, y \in Y} W_{xy} - \frac{vol(X)vol(Y)}{vol(G)} \right| \leq \bar{\lambda} \sqrt{vol(X)vol(Y)}$$

where $\bar{\lambda} = \underset{i \neq 1}{max} |1 - \eta_i|$. Here $\{\eta_i\}_{i \in [2..N]}$ are eigenvalues of the normalized Laplacian $\mathcal{L}(G) = D^{-\frac{1}{2}} L(G) D^{-\frac{1}{2}}$.

Ideally, this theorem could be applied directly to a σ-connected graph G to bound the difference between the edge weights of G and $\hat{G}$. In order to apply lemma 11, an approximation of $\bar{\lambda}$ is required.

LEMMA 12. *Assume graph G is coarsened to a single node and then lifted to $\hat{G}$. If $|\lambda_i - \hat{\lambda}_i| \leq \epsilon$ then $|\eta_i - \hat{\eta}_i| \leq min\{h_p, 1\}$ where $h_p = \frac{5p}{1-2p}$ and $p = \frac{\epsilon}{\delta} < \frac{1}{4}$.*

Proof. Begin by noting that the random-walk normalized Laplacian $\mathcal{L}_{rw} = D^{-\frac{1}{2}} \mathcal{L} D^{\frac{1}{2}} = D^{-1} L$ has the same eigenvalues as the normalized Laplacian. It is true from Lemma 10 that $|d_v - \delta| \leq 4\epsilon$. This implies the following bounds on the eigenvalues of D^{-1}.

$$\frac{1}{\delta + 4\epsilon} \leq \lambda_i(D^{-1}) \leq \frac{1}{\delta - 4\epsilon}$$

This implies that the nontrivial eigenvalues $\{\eta_i\}_{i \in [2..N]}$ of $D^{-1}L$ lie in the following bounds.

$$\begin{aligned} \frac{\delta - \epsilon}{\delta + 4\epsilon} \leq \eta_i \leq \frac{\delta + \epsilon}{\delta - 4\epsilon} \\ \Rightarrow \frac{1 - p}{1 + 4p} \leq \eta_i \leq \frac{1 + p}{1 - 4p} \\ \Rightarrow \frac{(1-p) - (1+4p)}{(1+4p)} \leq \eta_i - 1 \leq \frac{(1+p) - (1-4p)}{(1-4p)} \\ \Rightarrow \frac{-5p}{(1+4p)} \leq \eta_i - 1 \leq \frac{5p}{(1-4p)} \end{aligned}$$

For $0 \leq p < \frac{1}{4}$, $\frac{5p}{(1-4p)} > \frac{5p}{(1+4p)}$. Additionally, $\hat{\eta}_i = 1$ implying the following and completing the proof.

$$|\eta_i - \hat{\eta}_i| \leq min \left\{ \frac{5p}{1-4p}, 1 \right\}$$

□

Lemma 12 provides a bound on $\bar{\lambda}$ which, in conjunction with Lemma 11, may be used to prove that the differences between edge weights in G and $\hat{G}$ remain bounded within partitions. This bound does require $p = \frac{\epsilon}{\delta}$ to be rather small, however, since $\frac{5p}{1-4p} \to \infty$ as $p \to \frac{1}{4}$. In fact, $p = \frac{1}{9}$ is where this bound becomes devoid of useful information, since $|\eta_i - \hat{\eta}_i| \leq 1$ by virtue of this being a difference of normalized Laplacian eigenvalues. We can now prove a useful discrepancy bound for the case $p < \frac{1}{9}$.

Lemma 13. *Let there be a weighted graph $G = (V, W)$. Additionally, consider the lift of the graph $\hat{G} = (\hat{V}, \hat{W})$, which comes from first coarsening G to a single node. This implies $\hat{G}$ is δ-complete with $\delta = \frac{Vol(G)}{N}$. If $|\lambda_j - \hat{\lambda}_j| \leq \epsilon$ for all $j \in [1..N]$, then $|W_{uv} - \hat{W}_{uv}| \leq min\{h_p, 1\}(\delta + 4\epsilon) + \frac{4\epsilon}{N}(p+1)$ where $h_p = \frac{5p}{1-4p}$ and $p = \frac{\epsilon}{\delta} < \frac{1}{9}$.*

Proof. We directly apply Lemma 11 by considering $X = u$ to be a single node and $Y = v$ to be a single node.

$$\begin{aligned}\left|W_{uv} - \frac{d_u d_v}{vol(G)}\right| &\leq \bar{\lambda}\sqrt{d_u d_v} \\ &\leq \frac{5p}{1-4p}(\delta + 2\epsilon)\end{aligned}$$

The right hand side follows from Lemma 10 and Lemma 12. Going forward, $\frac{3p}{1-2p}$ will be denoted by h_p. Note that $vol(G) = vol(\hat{G}) = N\delta$.

$$\begin{aligned}\frac{\delta^2 - 4\delta\epsilon + 4\epsilon^2}{N\delta} \leq \frac{d_u d_v}{N\delta} &\leq \frac{\delta^2 + 4\delta\epsilon + 8\epsilon^2}{N\delta} \\ \Rightarrow \frac{4p\epsilon - 4\epsilon}{N} \leq \frac{d_u d_v}{N\delta} - \frac{\delta}{N} &\leq \frac{4p\epsilon + 4\epsilon}{N} \\ \Rightarrow -\frac{4p\epsilon + 4\epsilon}{N} \leq \frac{d_u d_v}{N\delta} - \frac{\delta}{N} &\leq \frac{4p\epsilon + 4\epsilon}{N} \\ \Rightarrow \left|\frac{d_u d_v}{N\delta} - \frac{\delta}{N}\right| &\leq \frac{4\epsilon}{N}(p+1)\end{aligned}$$

Using this, we can refine our statement further, thus proving our lemma.

$$\left|W_{uv} - \frac{\delta}{N}\right| = \left|W_{uv} - \hat{W}_{uv}\right| \leq min\{h_p, 1\}(\delta + 2\epsilon) + \frac{4\epsilon}{N}(p+1)$$

□

Using Lemma 13, the main theorem is ready to be proven.

THEOREM 14 (EDGE APPROXIMATION). *Let G be a σ-connected graph with respect to the partition $P = \{V_1, \cdots, V_k\}$, and let $\hat{G}$ be the lift of G with respect to that partition. Additionally, assume $|V_i|$ is large for each $i \in [1..k]$. If $|\lambda_i - \hat{\lambda}_i| \leq \epsilon$, the difference between in-partition weights is bounded by $|W_{uv} - \hat{W}_{uv}| \leq min\{h_q(j), 1\}\,(\delta(j) + 2(\epsilon + 2\sigma)) + \frac{4(\epsilon+2\sigma)}{N}\,(q(j)+1)$ for $u, v \in V_j$. Additionally, between-cluster weights are bounded by $|W_{uv} - \hat{W}_{uv}| \leq \sigma$ where $u \in V_i$ and $v \in V_j$ where $i \neq j$. Here, $h_q(j) = \frac{5q(j)}{1-4q(j)}$ and $q(j) = \frac{\epsilon+2\sigma}{\delta_j}$.*

Proof. From the definition of σ-connected, the Laplacian R is such that $\|R\|_1 \leq \sigma$. This bounds the maximum value of the matrix, implying that $|W_{uv} - \hat{W}_{uv}| \leq \sigma$ for $u \in V_i$ and $v \in V_j$ where $i \neq j$. For in-partition weights, first note that from Corollary 9, the eigenvalues of $C(j)$ and the eigenvalues of $\hat{C}(j)$ are bounded such that $|\mu_i(j) - \hat{\mu}_i(j)| \leq \epsilon + 2\sigma$. By using this as the error term in Lemma 13, one obtains the following bound: $|W_{uv} - \hat{W}_{uv}| \leq min\{h_q(j), 1\}\,(\delta(j) + 2(\epsilon + 2\sigma)) + \frac{4(\epsilon+2\sigma)}{N}\,(q(j)+1)$, which proves the theorem.

□

3.2.5 Discussion

Theorem 14 states that as the difference in spectrum $|\lambda_i - \hat{\lambda}_i|$ approaches zero for all $i \in [1..N]$, the difference in the weights of all edges depends only on the connectivity between subgraphs in the partition P. As a consequence of this, one can in a sense "hear" the shape of the original graph, given a coarsened graph G_c whose lift $\hat{G}$ spectrally approximates it. In practice, this bound is only practical for graphs which do not occur in general applications. To observe why, assume that a simple graph G is coarsened to G_c with respect to some partitioning $P = \{V_1, \cdots, V_k\}$. Further assume that within any partition V_i there are two nodes which are not adjacent. Because all nodes within the same partition are adjacent in the lift, the maximum edge-weight difference is bounded below by $\frac{\delta_i}{N}$. This minimum upper bound exists regardless of the spectral properties of G and $\hat{G}$. Furthermore, in most real world graphs, σ is relatively large and the resulting bound in Theorem 14 is dominated by the σ term in the expression, often leading to bounds larger than the largest degree in the graph. This implies that meaningful uses of the upper bound in Theorem 14 may generally

be restricted to weighted graphs where every node is adjacent to every other, and there are small weights between partitions. While not generally found in social-science, or scientific computing applications, such graphs are used in practice for image segmentation [50]–[52], and data mining tasks [29]. These methods use weighting schemes based on the distance between nodes to define similarities in arbitrary data. One common weight function is $w_{uv} = exp\{\|r_u - r_v\|_2^2/\Theta\}$ where r_u, r_v are the embeddings of u, v in $\mathbb{R}^N$, and Θ is a positive constant. Given this, or other similar weighting schemes, one can bound the distance between r_u and r_v. This implies that by bounding edge weights between the graph G and it's lift $\hat{G}$, one is simultaneously preserving the distances between these nodal embeddings. However, the bound in theorem 14 is only usable in the most well clustered of test cases, and requires further refinement before being usable in application.

3.3 Edge weight approximation for weighted regular graphs

We briefly turn our attention to a special case where the spectrum fully determines the properties of graph connectivity. This is in the case of weighted regular graphs where $d_i = d$ for all nodes $i \in V$ and some positive real number d. For this purpose we will instead examine the adjacency matrices W and $\hat{W}$. Coarsening as defined in definition 1 may be expressed as a matrix product $W = SWS^T$ for a coarsening matrix S discussed in further detail in Loukas [35]. Additionally the lifting operation can be expressed as the pseudo-inverse of this operation, given by $\hat{W} = P^\dagger PW(P^\dagger P)^T$. The matrix $PP^\dagger = \Pi$ has a simple form given in both Loukas [35] and Jin et al. [36]. Given a partition $P = \{V_1, \cdots, V_k\}$ each element $\Pi_{ij} = \frac{1}{|V_r|}$ for $i, j \in V_r$, otherwise $\Pi_{ij} = 0$. One can easily check that this coincides with our definition of coarsening.

This matrix relation between the original and lifted adjacencies allows for a powerful theorem to be proven.

THEOREM 15. *For a weighted adjacency matrix W and lifted adjacency matrix $\hat{W} = \Pi W \Pi$, if $|\omega_i - \hat{\omega}_i| \leq \gamma$ for all $i \in [1..N]$, then $\|W - \hat{W}\|_F^2 \leq N\gamma\,(2\|W\|_2 + \gamma)$ where $\|\cdot\|_F$ is the Frobenius norm.*

Proof. We begin by breaking the Frobenius norm into individual traces.

$$\begin{aligned}\|W - \Pi W \Pi\|_F^2 &= Tr((W - \Pi W \Pi)^2) \\ &= Tr(W^2) + Tr(\hat{W}^2) - 2Tr(W\hat{W}) = Tr(W^2) - Tr(\hat{W}^2)\end{aligned}$$

Additionally note the following.

$$\begin{aligned}|\omega_i - \hat{\omega}_i| &\leq \gamma \\ \Rightarrow |\omega_i + \hat{\omega}_i| &\leq \gamma + 2|\omega_i| \\ \Rightarrow |\omega_i^2 - \hat{\omega}_i^2| &\leq \gamma^2 + 2\gamma|\omega_i|\end{aligned}$$

From here, each trace is considered independently, with the intent of upper bounding $\|W - \Pi W \Pi\|_F^2$.

$$\begin{aligned}Tr(\hat{W}) - Tr(\hat{W}^2) &= \sum_{i \in V} (\omega_i^2 - \hat{\omega}_i^2) \\ &\leq \sum_{i \in V} (\gamma^2 + 2\gamma|\omega_i|) \\ \Rightarrow \|W - \hat{W}\|_F^2 &\leq N\gamma(\gamma + 2\|W\|_2)\end{aligned}$$

□

3.3.1 Discussion

Theorem 15 states that, preserving the spectrum of the adjacency matrix while coarsening is sufficient to preserve all edge weight information. This is a far stronger statement than the one proposed in Theorem 14. However, this is only applicable when the adjacency spectrum is preserved, not necessarily the Laplacian since the two spectra are not directly related for general graphs. In the case of weighted regular graphs it is easy to check that these are one in the same since, for a weighted regular graph with degree d, $\lambda_i = d - \omega_i$. Unfortunately this is not true for most graphs. Using this theorem 15 in the general case will require bounding $|\omega_i - \hat{\omega}_i| \leq f(\epsilon)$ for some function $f(\cdot)$ where $|\lambda_i - \lambda| \leq \epsilon$ for all $i \in [1..N]$. This remains an open problem.

3.3.2 Conclusion

The contributions of this manuscript have been twofold. A result originally derived by Jin et al. [36] was generalized to the case of the combinatorial Laplacian. We showed that, by using this result, one can closely preserve the spectrum of the graph Laplacian while performing graph coarsening. Additionally it was shown that the suggested coarsening criteria implies bounds on algebraic distances between nodes of the same graph. A comparison between coarsening methods was also presented. The latter half of the manuscript studied how closely the edge weights of a graph's lift approximate those of the original graph under an assumption that their Laplacian spectra are close. A sufficiently tight bound would guarantee that arbitrary data sets in $\mathbb{R}^n$ imbued with a graph structure could be coarsened while preserving their relative embeddings in $\mathbb{R}^n$. This is a novel question with potential applications to image segmentation and data mining. Unfortunately the bound proven relies on the connectivity of the graph and is unlikely to be useful in real world applications. It was then shown that, in the case of weighted regular graphs the connectivity of the graph does not require consideration, and a spectral approximation provides an edge weight approximation. Various avenues for extensions and branching research exist.

One obvious path for future research is to diminish the bound provided in Theorem 14. The proof for the theorem relies heavily on a discrepancy bound which is particularly loose. By circumventing this, perhaps with a more sophisticated extension to Lemma 10, one may be able to significantly tighten this bound. As an extension of this, removing the dependency on σ is important for applicability. In practice σ will be too large for this bound to be useful to practitioners. One avenue for exploring this may be to relate spectral differences between the adjacency and coarsened adjacency with those of the Laplacian and coarsened Laplacian, and then apply Theorem 15. Additionally, there are several interesting questions one may ask about the effects of coarsening arbitrary data sets. For instance, if the spectrum between a graph G and it's lift are close, how close are their edge weights on average? This is answered in a special case by Theorem 15, but is not known in general. This question is significantly less restrictive than the one presented in this section, however it still provides insight into the effects of coarsening on node embeddings. As for extending the results discussed in section 2, while it was shown that the coarsening criteria in Theorem 5 implies a bound on algebraic distances, a result in the opposite direction would be preferable. Algebraic distances are

cheap to compute for small numbers of test vectors, and if there were a reasonable guarantee on the accuracy of the spectrum, they would be preferable to the criteria presented in this thesis. Such a bound would likely be probabilistic for $k < N$ due to the fact that ensuring linear dependence between nodes using the algebraic distance requires the test vectors to span $\mathbb{R}^N$.

3.4 Introduction to the spectral coarsening implementation

Several methods exist for performing the task of spectrum preserving coarsening, however they are often computationally expensive. For instance in Liu et al. [53] a method is suggested where an initial combinatorial coarsening is performed and then cluster assignment is refined by iterating upon and optimizing a given matrix norm. This method does not have an explicit bound on its run time and instead iterates gradient descent to within some tolerance. Alternatively, Loukas [35] suggests a "local variation" method where k eigenvalues are pre-computed and then used as a point of comparison when considering potential merges. The explicit complexity of this method is given by $\tilde{O}(ckm + k^2n + ck^3 + \sum_{l=1}^{c} \Phi_l(min\{k^2\delta + k\delta^2, k\delta^2 + \delta^3\} + log|\mathcal{F}_l|))$. Here c is the number of coarsening levels, k is the number of desired eigenvectors matched, n is the number of nodes, and m is the number of edges in the original graph. For further details bout this complexity, one can visit the original publication. Both of these methods boast powerful spectral guarantees and impressive results. However, as discussed, they inherently have limitations in terms of compute time due to both the usage of expensive linear algebra operations as well as the methods being inherently sequential and unable to make use of modern parallel architectures. Other spectrum preserving coarsening methods [54], [55] fall victim to similar issues. In this section we aim to reduce the time required to perform spectrum consistent coarsening. We derive a new spectral approximation bound for agglommerative coarsening and implement our algorithm on GPU.

3.5 Greedy algorithm

In [36] a greedy algorithm for spectrum consistent coarsening is defined. This algorithm, which has a complexity of $O(m(n+n_c)(n-n_c))$ will be called the "explicit greedy algorithm" for the remainder of this section. The algorithm works as follows. For every edge in the graph the 1-norm is computed between the adjacency vectors of the nodes at the ends of that edge. Then these 1-norms are sorted and a merge is performed along the edge with the smallest 1-norm. These steps are then repeated to our desired level of coarsening. First we note how the inner two loops of the explicit greedy algorithm may be parallelized. These loops iterate over each of m edges and compute the associated norm-difference between node adjacencies. Therefore we are computing $O(m)$ norms, where the computation of each norm between nodes $u, v \in V$ has sparse vector work complexity $O(d_u + d_v)$. This gives us an overall complexity for these inner loops of $O(n \langle d^2 \rangle)$ where $\langle d^2 \rangle$ is the second moment of the degree distribution of our graph. Note that none of these norms depend on previously computed norms, so these inner loops can be parallelized such that given p available threads, if we evenly distribute the edges among threads we achieve a shared-memory parallel-time complexity of $O(\frac{n}{p} \langle d^2 \rangle)$ for our inner loops, where p is the number of threads.

To achieve further complexity reduction we would like to be able to remove the outer loop of the greedy algorithm. This will allow us to avoid recomputing norm differences. For this purpose we make an additional observation that the eigenvalue differences of an arbitrary level of coarsening may be bounded in terms of the norm differences in the original uncoarsened graph given by the following theorem.

THEOREM 16. *Given a set of s merges $\{(a_1, b_1), \cdots, (a_s, b_s)\}$ where $\|\frac{w_{a_i}}{d_{a_i}} - \frac{w_{b_i}}{d_{b_i}}\|_1 \leq \epsilon$ for every $i \in [1..s]$, then we know $\|\Lambda - \hat{\Lambda}\|_\infty \leq \frac{s(s+1)}{2}\epsilon$.*

Proof. The proof breaks in to two parts. First we wish to show that $\|\frac{w_a}{d_a} - \frac{w_b}{d_b}\|_1 \leq \epsilon$ implies, without loss of generality, that $\|\frac{w_a}{d_a} - \frac{w_a+w_b}{d_a+d_b}\|_1 \leq epsilon$. This statement says that if two nodes have a small adjacency difference norm before merging, each of them will also have a small difference norm when compared with the merged node. Take $v = \frac{w_a}{d_a} - \frac{w_a+w_b}{d_a+d_b}$, then by algebra it can be shown that $\|v\|_1 \leq \frac{\epsilon}{\frac{d_a}{d_b}+1} \leq \epsilon$, which directly implies $\|\frac{w_a}{d_a} - \frac{w_a+w_b}{d_a+d_b}\|_1 \leq \epsilon$

For the second half of our proof, imagine we have performed some number of merges.

Then there are two possibilities for the subsequent merge. Either it shares no nodes with the previous merge, or it shares nodes with the previous merge. In the first case, neither of the nodes have been coarsened, so their similarity remains the same and the specral approximation theorem from Jin and Loukas [36] tells us that we simply add ϵ to our spectral error bound after merging them. In the latter case, assume without loss of generality that our merge consists of nodes a, c where a overlaps with the first merge. Then by triangle inequality, and the first part of our proof we know $\|\frac{w_c}{d_c} - \frac{w_a+w_b}{d_a+d_b}\|_1 \leq 2\epsilon$. This means that the normalized adjacency vector of node c is 2ϵ similar to the normalized adjacency vector of the supernode given by our first merge. Therefore we can apply the Jin and Loukas bound again and this error of 2ϵ, adds to our bound. We may then repeat this process until our desired level of coarsening, adding a factor of ϵ to each addition. The worst case error occurs when all the merges overlap, in which case the upper bound is given by the following.

$$\|\Lambda - \hat{\Lambda}\|_\infty \leq \sum_{k=1}^{s} k\epsilon = \frac{s(s+1)}{2}\epsilon$$

□

While looser than the bound provided in [36], this bound requires no knowledge about intermediate coarsenings of our graph. Because of this, we can remove the outer loop of the explicit greedy algorithm to obtain an order $O(n)$ work complexity reduction while still preserving a spectral bound. This gives us our Algorithm 4.

Algorithm 4 Approximate Greedy Coarsen (G = (V,E), n_c, p)

```
s ← |V|
fitness ← ∅
{E_1, ··· , E_p} ← partition(E, p)
for (a, b) ∈ E_i in parallel do
    fitness ← fitness ∪ ((a, b), ‖w_a/d_a − w_b/d_b‖_1)
fitness ← sort(fitness)
while s < n_c do
    (a, b) ← fitness[index][0]
    G ← merge(a, b)
    s ← s − 1
```

Algorithm 4 closely resembles the explicit greedy algorithm with some reorganization. We still iterate through every edge and calculate a fitness function which is the norm-difference of the normalized adjacencies between each node, however we only perform this

once for each edge. Then a sort is performed, which can be done in $O(\frac{m}{p}log(m))$ parallel time. Finally, our merges are performed in order of fitness, requiring $O(n - n_c)$ operations. This gives us a final parallel time of $O(\frac{n}{p}\langle d^2 \rangle + \frac{m}{p}log(m) + (n - n_c))$ for Algorithm 4.

3.6 Results

We present results for both the scaling and approximation properties of our algorithm on two small example graphs, the "Stanford bunny" from the Stanford 3D Scanning Repository as well as "ego-Facebook" from the Stanford Large Network Dataset Collection. The former is a nearly-regular mesh and the latter has a highly irregular degree distribution. We find that while spectral approximations appear better on the irregular "ego-Facebook", the parallel time scaling is far superior on the mesh.

3.6.1 Scaling

We observe in Figure 3.5 the runtime scaling of our algorithm for the "ego-Facebook" and "Stanford bunny" graphs. We can see that, the Facebook graph has drastically worse scaling properties which level out well before reaching the final 2048 thread test. Additionally the Facebook graph requires far more time to coarsen, despite it being almost a tenth the size of the bunny mesh. In comparison, the bunny mesh experiences nearly theoretically ideal strong scaling. This scaling difference can be attributed to load imbalance. Note that in our parallel algorithm we do not parallelize within each norm computation. For a highly irregular network such as "ego-Facebook" algorithm 4 partitions edges naively across threads without any consideration for the nodes at either end of each edge. The irregularity of the degree distribution in "ego-Facebook" guarantees that norm-differences along certain edges will require far more compute time than others, and by not accounting for this it is likely that some threads have significant work to perform when computing norms, while others have little to compute. As an example, in the norm computation step one thread may own k edges which connect to the largest degree node in the network with degree D, while another may own k edges that all connect to nodes with unit degree. In this case the latter

http://graphics.stanford.edu/data/3Dscanrep/
https://snap.stanford.edu/data/

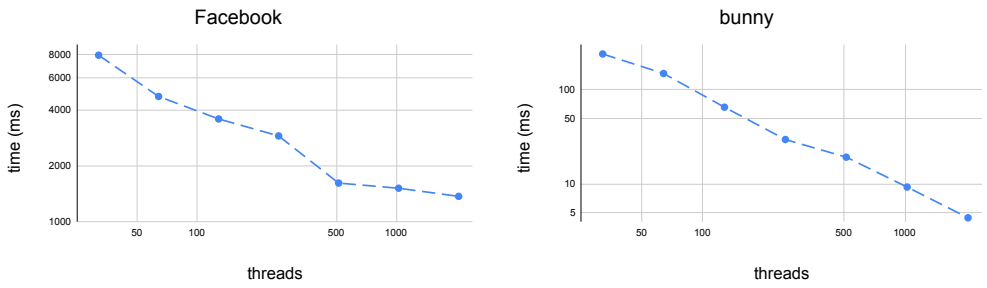


Figure 3.5: Scaling for two different topologies. We compare the scaling properties of our algorithm running from 32 to 2048 threads on a single Nvidia Quatro M5000 on each of the two test graphs. We can see that the runtimes on the bunny mesh is incredibly small in comparison with the Facebook graph, which has a highly irregular degree distribution. Also notably, we observe that the bunny mesh almost reaches theoretical perfect scaling.

thread only has to complete $2k$ operations, while the former thread has to complete more than $k(D+1)$ operations. This can be mitigated in future work through more sophisticated thread parallelism and usage of more modern GPU architectures, where warp divergence of threads is less of an issue [56].

3.6.2 Spectral approximation

In Figure 3.6 we present the eigenvector and eigenvalue approximation properties for the "Stanford bunny" as well as "ego-Facebook" on the first fifty nontrivial eigen-pairs. We compare the approximation properties on graphs coarsened to half the number of nodes, quarter the number of nodes, and an eighth the number of nodes. We can see that for both graphs, the half-coarsening is a reasonably accurate approximation of the original eigenvalues, however the eigenvalues quickly stray as we add additional levels of coarsening. As for eigenvector approximations, we consider the inner product between the first fifty normalized eigenvectors of the original graph with the first fifty normalized eigenvectors of the lifted graph for each level of coarsening. In Figure 3.6 eigenvector comparisons comprise the rightmost three columns. The better the eigenvector approximation is, the more each matrix will resemble an identity matrix since the eigenvectors of the normalized Laplacian are orthogonal [37]. We see that the eigenvector approximation appears better for "ego-Facebook" and for the "Stanford bunny" becomes rather poor after half coarsening.

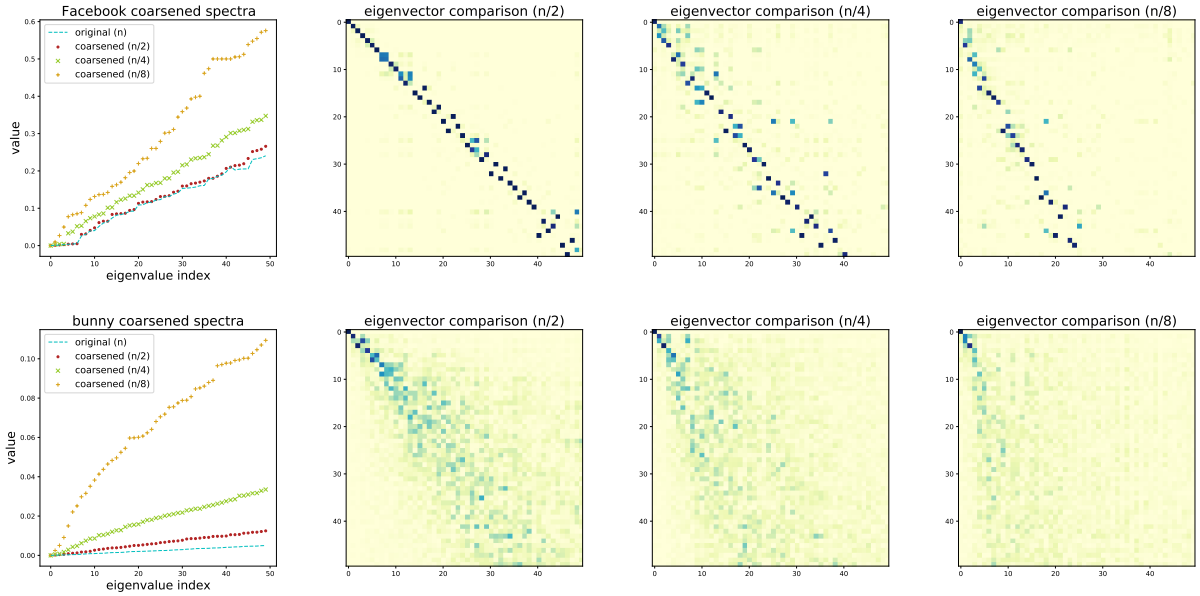


Figure 3.6: Spectral and eigenvector approximation. We present spectral approximation properties on the two test-graphs. Each graph was coarsened to half size, quarter size, and eighth size. On the left we compare the first 50 eigenvalues of each lift against the eigenvalues of the original graph. The right three columns compare the angles between the associated eigenvectors by considering the dot product of the eigenvectors in the original graph, with the eigenvectors of each lifted graph. Intuitively, the closer this matrix resembles the identity, the better the eigenvector approximation is.

We also show example eigenvectors on the bunny graph and the lift of the half-coarsened bunny graph in Figure 3.7. We can see that, while the positive and negative regions drift for the selected eigenvectors, the overall pattern is somewhat conserved. This is of significant interest since the sign patterns of the eigenvectors are important for tasks such as spectral clustering [29].

3.6.3 Conclusion

In this section we iterated upon a previously researched algorithm for coarsening graphs while preserving approximate eigenvalues. This method can be used to quickly compute spectrally approximate coarsenings of general graph data such as geometric and finite element meshes. We presented a new bound for the eigenvalue differences between original and coarsened graphs, and presented a parallel algorithm for performing graph coarsening within

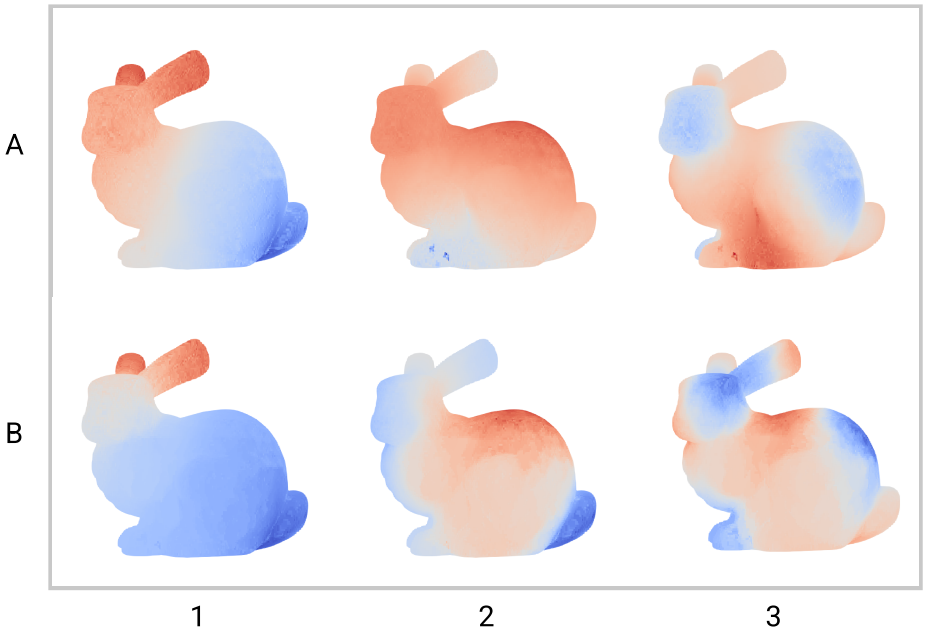


Figure 3.7: Bunny eigenvectors. Here we plot three eigenvectors on the original bunny mesh as well as the corresponding lifted eigenvectors on the half-coarsened bunny mesh. The A row are the eigenvectors on the original mesh, and the B row contains the eigenvectors of the coarsened mesh. From left to right the columns correspond to the Fiedler vector, the fifth, and the tenth nontrivial eigenvectors respectively. The blue portions denote negative values and red portions denote positive values. We can see that the eigenvector values have drifted even though the eigenvalues are reasonably close as is seen in Figure 3.6.

this bound. Our suggested algorithm has a parallel time of $O(\frac{n}{p}\langle d^2\rangle + \frac{m}{p}log(m) + (n - n_c))$ for coarsening graph $G = (V, E)$ to n_c nodes, a vast theoretical speed up over other previously suggested coarsening algorithms preserving spectra. We additionally showed that our algorithm remains faithful to the spectrum of the original graph for limited amounts of coarsening, however the spectrum deviates significantly as the number of nodes in the coarsening diminishes.

Our proposed coarsening algorithm leaves room for improvement in the areas of accuracy and parallel time scaling. For the former, we observe that our Bound 16 is rather loose, and it may be tightened by recomputing difference-norms between nodes after some amount of coarsening. This suggests that greater overall accuracy may be obtainable by iterating the difference-norm computations after some amount of coarsening. The optimal amount of

coarsening between such iterations remains an open problem. For parallel scaling, several improvements could be made. As mentioned earlier in the chapter, we partition edges naively among threads which leads to load balancing issues for irregular degree distributions. This could be remedied by considering more sophisticated methods for partitioning the edge set, however this will likely come at the cost of additional preemptive computations. Additionally, faster norm-computations may be achieved by using finer-grained parallelism, where multiple threads are assigned to compute each individual norm. Currently, each norm is computed sequentially by a single thread, which leads to warp divergence and compounds our imbalance issues. Well-known techniques such as hierarchical parallelism [57], loop collapse [58], and graph adjacency reordering [59] are relatively straightforward to implement and will greatly improve speedups for irregular graph datasets.

3.7 Extensions

Graph coarsening is a topic which has received a great deal of study due to its use in multilevel schemes for numerical algorithms. Despite this, little work has been done on the inverse problem, and there is a significant amount of room to expand upon the topic. The bounds given in Theorem 14 are incredibly loose in general, and need to be tightened for practitioners. This may potentially be done by exploring specific topological constraints as in Theorem 15, or other proof techniques not explored here. There are a few reasons such bounds may be useful in practice. Assume that we wish to solve some large, symmetric linear system defined by the matrix W. Further assume that we have some bound such that $\Pi W \Pi = W + E$ for a coarsening defined by P, where $\Pi = P^{\dagger}P$. Note that chapter three focuses on bounding the norm $\|E\|$ when the difference in eigenvalues between W and $\Pi W \Pi$ is small. In this case we can ensure the following for least-squares solutions of the linear

system.

$$\Pi W \Pi a = b + \theta \tag{3.7}$$

$$\Rightarrow (W + E)\, a = b + \theta \tag{3.8}$$

$$\Rightarrow Wa - b = Ea + \theta \tag{3.9}$$

$$\Rightarrow \|Wa - b\| = \|Ea + \theta\| \tag{3.10}$$

$$\Rightarrow \|Wa - b\| \leq \|E\|\|a\| + \|\theta\| \tag{3.11}$$

This says that least squares solutions for linear equations involving $\Pi W \Pi$ yield errors with respect to W which are bounded by a combination of the per-weight errors and the quality of the original least squares approximation. But, we also have to remember that $\Pi W \Pi$ can be perfectly recovered from the coarsened matrix $W_c = PWP^T$, since $\Pi W \Pi = P^\dagger PW(P^\dagger P)^T = P^\dagger W_c (P^\dagger)^T$. This means such a solution a in equation 3.11 can also be found by solving the simplified system on W_c as follows.

$$W_c (P^\dagger)^T a = b' + \theta' \tag{3.12}$$

$$\Rightarrow \Pi W \Pi a = P^\dagger b' + P^\dagger \theta' \tag{3.13}$$

$$\therefore b = P^\dagger b' + \beta \Rightarrow \Pi W \Pi a - b = \beta + P^\dagger \theta' \tag{3.14}$$

$$\therefore \|Wa - b\| \leq \|\beta\| + \|P^\dagger \theta'\| + \|E\|\|a\| \tag{3.15}$$

Equation 3.15 yields a way to bound the error of approximate solutions to linear systems in W given by solving a system using the smaller matrix W_c. In this case we solve for $a' = (P^\dagger)^T a$ on the coarsened level, and then the equivalent solution $a = P^T a'$ is used as the approximate solution for the original system. If $b \in \text{span}(\Pi)$, this means the solution error depends entirely on the solution quality on the coarsened graph, and the edge approximation quality defined by the matrix E. This equation provides a useful bound which may be used to predict the quality of multilevel schemes for solving linear systems. Because of this, finding coarsenings for which $\|E\|$ is minimized presents a compelling avenue for further research.

CHAPTER 4
TRAINING GCN WITH KOOPMAN OPERATORS

4.1 Chapter overview

In this chapter we discuss the acceleration of graph neural networks for node classification. We investigate how to accelerate backpropogation by interweaving approximation steps with standard backpropagation steps to avoid explicit computations. For this purpose we utilise Koopman theory, and apply these approximations on GPU.

4.2 Introduction

Neural networks are ubiquitous in many domains of study including physics, biology, and general data science. Because of this, accelerating the training time of neural networks is an incredibly active area of research. Graph neural networks (GNN) and more specifically, graph convolutional networks (GCN) are of particular contemporary interest. Since deep neural networks (DNN), convolutional neural networks (CNN), and recurrent neural networks (RNN) require ordered data in order to perform well, the ability of GCNs to perform tasks on unordered data presents an avenue for learning from datasets which have been historically cumbersome or infeasible. Due to the rising interest in GNNs and GCNs, many methods for accelerating training have been proposed (see [60]–[62] for recent surveys). These methods include graph sparsification, graph coarsening, and most notably graph pooling. The concept behind each of these methods is to alter the graph topology during training in order to resolve issues with the "neighborhood blow-up" associated with many real world networks. While these methods are topological simplifications in nature, GNNs use the same standard backpropagation-based optimizers as traditional neural network architectures. It then makes

Portions of this chapter have been submitted as: C. Brissette, W. Hawkins, and G. M. Slota, "Acceleration of GNN node classification using Koopman operator theory on GPU.", *Adv. Neural Inf. Process. Syst.*

sense that acceleration in optimizers for traditional neural networks may lead to acceleration in GNNs. For this purpose, we introduce the Koopman operator ([63]).

The Koopman operator is of historical importance in the field of functional analysis. Originally introduced by Bernard Koopman in 1931, it provides a method for analyzing finite dimensional nonlinear dynamics with an infinite dimensional linear system. While the study of this operator was broadly relegated to the whiteboards of functional analysts for many decades, it has experienced a renaissance in the applied community in the last decade ([64]–[66]). This is largely due to the popularity of a method from computational fluid dynamics known as the dynamic mode decomposition (DMD) being interpreted as a finite dimensional approximation of the Koopman operator ([67]).

The utility of the Koopman operator is multifaceted, but for the case of neural network training it is important for one main reason. The discrete Koopman operator allows for accurate predictions in the state evolution of a nonlinear dynamical system at the cost of a single matrix-vector multiplication. Previous work ([68], [69]) has made this connection and shown that by treating the weights of a neural network as the state variables in a nonlinear dynamical system, Koopman operator theory and DMD may be used to train neural networks. The prior work has also shown that the weights output by Koopman training approximate those of a network fully trained with the underlying optimizer. Despite this, previous publications on the subject either lack the speed to be useful to practitioners, or they are only shown to be useful near the optimal solution of a DNN. Furthermore, prior work focuses solely on DNN architectures and do not test on GCNs. We address both of these problems in this chapter.

4.2.1 Our contribution

We present a fast method for Koopman training entirely on GPU which performs well in terms of performance acceleration and memory requirements. We apply this method to three standard node-classification problems using GCNs and show that Koopman training can produce accurate results in this domain with much faster training times relative to Adam. We additionally discuss best practices to avoid and mitigate potential instabilities in these methods.

4.3 Background

4.3.1 GNNs / GCNs

GNNs, and more specifically GCNs, rose to popularity in the machine learning community after the publication of a popular paper by [70] detailing a convolutional architecture for learning on graphs. The Kipf and Welling architecture remains popular, and it will be the architecture referred to interchangeably by the acronyms GNN and GCN for the remainder of this chapter.

The fundamental idea behind GCNs, and specifically GCNs for node classification, is to produce embeddings for each node based on aggregates of neighboring feature vectors. The difficulty in this technique is learning the aggregation function for each neighborhood. For this purpose, Kipf and Welling suggested the following architecture.

$$H^{l+1} = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{l}W^{l}\right) \tag{4.1}$$

Here H^l is the matrix of activation functions at the l^{th} layer and $\hat{A} = A + I_{|V|}$ is the adjacency matrix of the underlying graph $G = (V, E)$ with added self-loops. Additionally, $\hat{D}$ is the diagonal degree matrix given by the sums of rows of $\hat{A}$, σ is an element-wise activation function, and W^l is a learnable matrix of weight parameters for layer l. For a node $u \in V$, this function considers a weighted sum of the activations H^l of its neighbors $v \in \mathcal{N}(u)$, multiplies that by a weight matrix W^l, and applies an element-wise activation function. The goal of training the GCN is to solve for all weights, W^l in each layer.

Without additional acceleration techniques, this method can be particularly slow. To deal with this, topological simplification methods such as graph pooling are utilized. There are many methods for graph pooling; however, the common idea is to sample from a reduced graph for training instead of using the entire graph topology. This is an active area of research, and techniques range from explicit spectral methods to heuristics ([71]–[74]).

4.3.2 Koopman operator

There is a wealth of information about the Koopman operator that is not required to understand the rest of this thesis. We present a less general definition of the Koopman operator

for our application. Denote by $x_t \in \mathbb{R}^n$, the state of a dynamical system at time $t \in \mathbb{R}$. Additionally, take F to define our dynamical system such that $F(x_t) = x_{t+1}$. Also consider any observable $g : \mathbb{R}^n \to \mathbb{R}$ on the Hilbert space $L^2(\mathbb{R}^n, \mu)$. The Koopman operator K is the infinite dimensional linear operator such that the following is true for all such functionals g.

$$Kg(x_t) = g(F(x_t)) \tag{4.2}$$

This means K is a linear operator which preserves all measurements of our dynamical system at time $t + 1$, given its state at time t. Because of this, it may be used to predict future states. Since we do not have access to storage for infinite dimensional matrices, practitioners who wish to use Koopman operator theory must rely on finite rank approximations of K. These finite rank approximations may take several forms, including the finite section method and the dynamic mode decomposition. While the dynamic mode decomposition is noteworthy and its variants have spawned a wealth of research, it requires the computation of eigenpairs for potentially non-symmetric matrices, and it is therefore not amenable to fast parallelism on GPU. For this reason, we will be focusing on the finite section method for Koopman approximation.

The finite section method is incredibly simple. Given a set of observations of the dynamical system as a matrix $x = [x_0, x_1, \cdots, x_T]$, they may be broken into two matrices $X = [x_0, x_1, \cdots, x_{T-1}]$ and $Y = [x_1, x_2, \cdots, x_T]$. These matrices define the "before and after" states, where we know $x_{t+1} = F(x_t)$. Then, the finite section approximation U of K is given by the following where $\dagger$ denotes the pseudo-inverse of a matrix.

$$U = YX^{\dagger} \tag{4.3}$$

From this, future states may be predicted using the equation $U^s x_t \approx x_{t+s}$. The intuition behind this method is that, by properties of the pseudo-inverse, $\|Ux_t - x_{t+1}\|_2$ is minimized for all (x_t, x_{t+1}) in x. This means that on our observed subspace we match the nonlinear dynamics as closely as possible with respect to the $L2$-norm.

4.3.3 Koopman training

The concept of Koopman training is simple: treat the weights in the neural network as the variables in a dynamical system and apply a Koopman operator at some time step to avoid back-propagation. Functionally, this amounts to tracking the weights of the neural network for some number of time steps m, then solving for a finite dimensional Koopman operator approximation U, and finally predicting p steps ahead using U^p. The advantage to this method is that matrix-vector multiplication is much faster than back-propagation.

We discuss both of the former works studying Koopman training of neural networks by [68] as well as the work by [69]. These papers, posted to the arXiv within a couple weeks of each other, present very different approaches to Koopman training. In the paper by Dogra and Redman, Koopman training is applied near the end of optimization when weight evolution is slow. Starting at some time t_1 they begin tracking weights until another time t_2. These weights are then used to obtain U via the finite section method. U is then used to approximate the final state of the network after many steps. In the paper, they predict an impressive 2500 steps ahead and show that their final network weights are very close to those of the traditionally trained network.

Tano et al. take a very different approach. Primarily, their method utilizes the dynamic mode decomposition as opposed to finite sections. This has two effects. First, it makes their method slower than the work of Dogra and Redman, since now an eigen-decomposition must be computed on top of the pseudo-inverse. Second, it additionally can provide improved accuracy, since DMD allows for the pruning of spurious modes which may create instabilities in predictions. Beyond that, the authors do not wait to use their Koopman operator as Dogra and Redman do. Instead, they alternate during training, performing m steps of standard optimization before predicting p steps forward in time and returning to standard training for another m steps. This allows for acceleration throughout the entire training process, not just near a minimum.

Both methods are shown to train their test networks well, however there are gaps for further research. As noted by both groups, these implementations are CPU only, but they require a GPU implementation to be useful to practitioners. In fact, Tano et al. note that their DMD-based method is ultimately slower than standard optimization. Furthermore all test instances in both papers were on DNN architectures, and it is yet to be shown how

similar methods may generalize to learning tasks such as GCNs.

4.4 Methods

4.4.1 Algorithm

Table 4.1: Variables for Koopman training (Algorithm 5) and prediction (Algorithm 6).

Variable	Definition	Experimental Values
m	Number of standard training steps	{4, 8, 16, 32}
p	Number of steps predicted forward	{4, 8, 16, 32}
r	Finite dimension for SVD truncation	{1, 2, 3, 4, 5}
M	Input network	–
ω	Number of learnable parameters in the network	–
W	Weight history matrix for prior steps	–

Our Koopman training algorithm takes place in two distinct alternating phases, as seen in Algorithm 5. In one phase (Line 8), training is performed using a standard optimization technique, such as stochastic gradient descent (SGD) ([75], or adaptive moment estimation (Adam) ([76]). This training is performed on some input network M for some number of pre-determined steps m, and the weights of the network are stored as a column $W_i \in \mathbb{R}^\omega$ in the weight history matrix $W \in \mathbb{R}^{\omega \times m}$, where ω is the number of learnable parameters in the network. We additionally use parameters r and p, which are the finite dimension for SVD truncation and number of steps predicted forward, respectively. We summarize the variables for our algorithms in Table 4.1.

Algorithm 5 : Koopman training $(M, m, p, r, \texttt{iters})$

1: $\omega \leftarrow \texttt{param\_num}(M)$
2: $W \leftarrow \texttt{zeros}(\mathbb{R}^{\omega \times m})$
3: $W \leftarrow \texttt{pad}(W, \omega \bmod 32)$
4: **for** $i \in [1..\texttt{iters}]$ **do**
5: **if** $i \bmod m = 0$ **then**
6: $\texttt{weights}(M) \leftarrow \texttt{koopman\_prediction}(W, m, p, r)$
7: **else**
8: $\texttt{weights}(M) \leftarrow \texttt{train\_epoch(M)}$
9: $W_{(i \bmod m)} \leftarrow \texttt{weights}(M)$
10: **return** M

In the second phase of Koopman training, seen in Algorithm 6 (and called from Line 6 in Algorithm 5), the weight history matrix is used to form the matrices $X = [W_0, W_1, \cdots, W_{m-1}]$ and $Y = [W_1, W_2, \cdots, W_m]$. Then the singular value decomposition (SVD) is computed for $X = Z\Sigma V^*$ and its pseudo-inverse is computed from that decomposition. This yields our finite section approximation of the Koopman operator U as the following.

$$U = YV\Sigma^{-1}Z^* \tag{4.4}$$

It should be noted that computing the SVD is an incredibly expensive operation with a work complexity of $O\left((m-1)^2\omega\right)$. One may notice that Algorithm 6 reshapes the data before performing SVD. This is because we subdivide both X and Y into small matrices for efficient batched computations on GPU. This yields a piece-wise approximation of U across subsets of weights in the network. In the language of Dogra and Redman, this is a type of sub-node level Koopman operator. We call this "patchwork Koopman training". Patchwork Koopman training is discussed in further detail in the 'Implementation' section that follows.

After U is computed, training is projected forward a pre-determined number of steps p by the equation $U^p W_m = W_{m+p}$. Afterwards, the weight history matrix is cleared, W_{m+p} replaces the old vector W_0, and we return to standard optimization for another m steps before repeating the process again. This is repeated for some number of iterations, or some early stopping criteria.

Algorithm 6 : Koopman prediction (W, m, p, r)

$X \leftarrow \texttt{reshape}\left([W_0, W_1, \cdots, W_{m-1}]\right)$
$Y \leftarrow \texttt{reshape}\left([W_1, W_2, \cdots, W_m]\right)$
$(Z, \Sigma, V^*) \leftarrow \texttt{batched\_svd}(X)$
$Z \leftarrow \texttt{trunc}(Z, r)$
$\Sigma \leftarrow \texttt{trunc}(\Sigma, r)$
$V^* \leftarrow \texttt{trunc}(V^*, r)$
$U \leftarrow YV\Sigma^{-1}Z^*$
$v \leftarrow \texttt{reshape}(W_m)$
for $j \in [1..p]$ **do**
    $v \leftarrow Uv$
return v

4.4.2 Implementation

A number of optimizations are made to the base Algorithm 5 in order to improve its performance on GPU. For starters, the expensive $O\left((m-1)^2\omega\right)$ SVD is broken into many subproblems. In Figure 4.1, it can be seen that Algorithm 6 reshapes the first $m-1$ columns of W into matrices of size $32 \times (m-1)$. This is done in order to make use of the batched SVD operation available via CUDA's gesvdjBatched function. This function allows for multiple singular value decompositions to be performed at once, so long as the shape of each individual matrix is smaller than or equal to 32×32 and all sub-matrices are the same size. Because of this, U, in our case, does not stand for the Koopman operator of the entire model. Instead, U can be thought of as a patchwork Koopman operator of smaller matrices $U = \{U_i\}$, where each matrix is an approximation of the full Koopman operator on a subspace with maximum dimension 32. In order to meet these requirements, zeros may need to be added in order to pad the size of W such that its column number is divisible by 32. This can be seen in Algorithm 5 as the function $pad(\cdot)$.

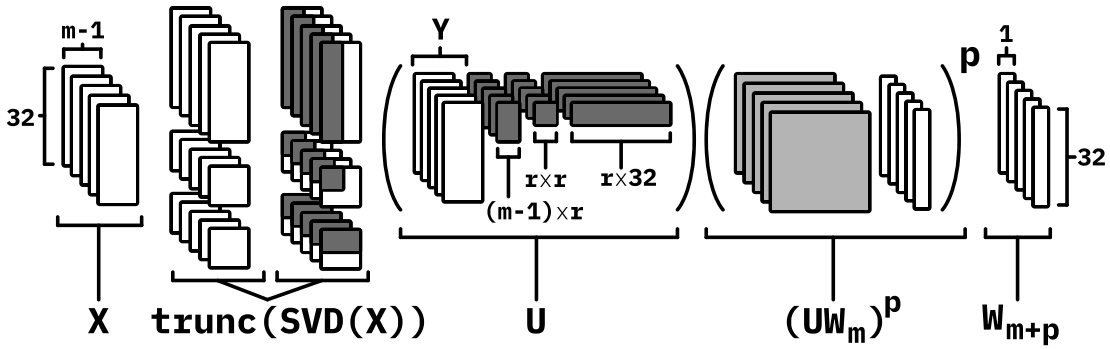


Figure 4.1: Visualization of Algorithm 6.

A further optimization is the truncation operation. Here, a finite dimension $r \leq m$ is chosen, and the SVD (Z, Σ, V^*) is truncated to that dimension. This is akin to using principal component analysis (PCA) ([77]) to reduce the dimensionality of X before computing the pseudo-inverse. This has two benefits. First, for $r < m$, this can drastically reduce both the total work required as well as the per-thread work. Additionally, for small r, this prunes minuscule singular values from Σ. If not pruned, these values may cause instabilities in the method. After the truncation step, $U = YV\Sigma^{-1}Z^*$ is computed. The influence of r on our patchwork Koopman implementation is discussed in the 'Best Practices for Mitigating Instabilities' section in the Discussion. Unfortunately, there are no readily available methods for batched eigen-pair computations of non-symmetric matrices on GPU. This relegates us to using the finite section method, and it means that fast implementations of more robust Koopman approximations such as Exact DMD ([78]) and ResDMD ([79]) are currently infeasible.

All algorithms were implemented using the PyTorch ([80]) and CuPy ([81]) libraries in Python. CuPy is GPU-optimized implementation of the functionality contained within NumPy and SciPy. While both of these libraries have access to functions utilizing gesvdjBatched, we used the PyTorch batched SVD function and *torch.linalg.svd*$(\cdot)$. CuPy was used for all other linear algebraic operations.

4.4.3 Experiments

We performed a parametric study for the task of GCN node classification over the parameters m, p, and r, which were varied over $\{4, 8, 16, 32\}$, $\{4, 8, 16, 32\}$, and $\{1, 2, 3, 4, 5\}$, respectively.

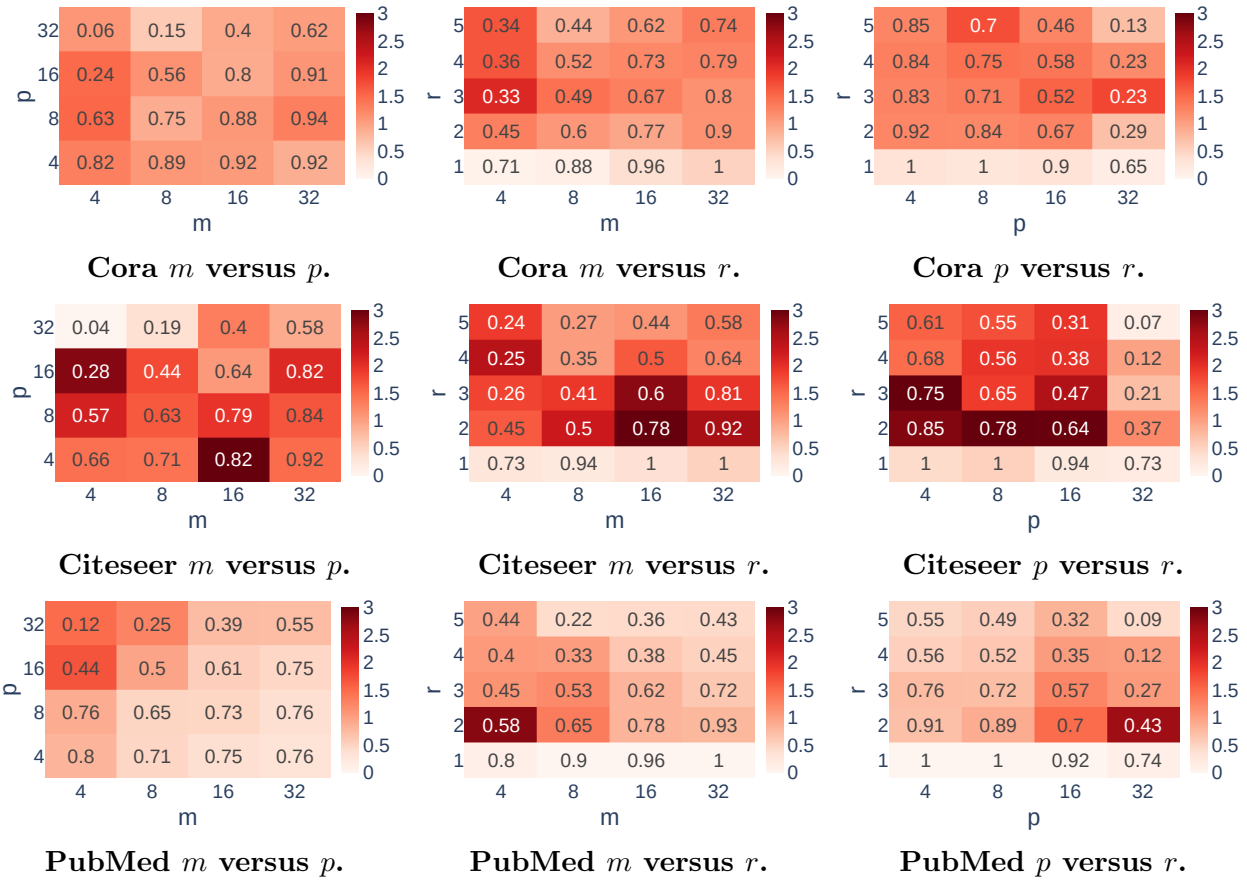
The parameters for m had to be less than 32 to allow use of batched SVD with CUDA. We also found in prior experiments that $p > m$ was susceptible to instability. Hence, we use the same parameter space for m and p. We limit $r \leq 5$, as experiments have demonstrated that higher r values result in increasing instability and lower speedups.

Each test consisted of 10 runs. All tests were performed on a single Nvidia Quadro M5000 GPU on the publicly available Cora ([82]), PubMed, and Citeseer ([83]) datasets. These three datasets are referred to as the Planetoid ([84]) collection in the PyTorch Geometric Library. This collection has been used extensively in GNN research for benchmarking node classification results (e.g., [70], [85], [86]). Two network architectures were considered. A GNN with a single convolution, and another with two convolutions. The outputs of all convolution layers were size 64, their activation function was ReLu, and the output was put through a softmax layer. All training was performed the Adam optimizer in PyTorch. These network architectures were selected because the weight numbers would be easily divisible for patchwork Koopman, and in testing they were expressive enough to train well for our classification problems.

4.5 Results

We examine the effects of m, p, r on the loss and accuracy of networks trained using Adam as well as our patchwork Koopman Adam (PKA). The results of this parametric study can be seen in Figure 4.2, and Figure 4.3. In these figures, networks are trained for 400 epochs using Adam, as well as being trained for the same duration using PKA. The speedup is then computed as $T(\text{Adam})/T(\text{PKA})$, where $T(\text{Adam})$ is the amount of time required for Adam to achieve its highest accuracy (maxpoint) or lowest loss (minpoint) over the 400 epochs, respectively, and $T(\text{PKA})$ is the amount of time required for PKA to achieve the same accuracy or loss. Instances where PKA does not achieve the desired accuracy, or where the loss explodes due to instability are omitted from the average. The proportion of experiments for a given parameter-pair which are stable is reported as a decimal value over each heatmap element in Figures 4.2 and 4.3.

In Figure 4.4 we show how the results from the heatmaps may be used to determine best-parameters in the case of the Cora dataset. In this instance, the earlier Figures 4.2 and 4.3 suggest $m = 4, p = 4, r = 2$ are good candidate parameters for accelerating the training



Cora m versus p. **Cora m versus r.** **Cora p versus r.**

Citeseer m versus p. **Citeseer m versus r.** **Citeseer p versus r.**

PubMed m versus p. **PubMed m versus r.** **PubMed p versus r.**

Figure 4.2: Maxpoint accuracy speedup. The average maxpoint accuracy speedup is shown for each pair of parameters (m, p), (m, r), (p, r), and each dataset Cora, Citeseer, and PubMed. Darker coloring implies greater speedup for our method. The proportion of stable runs is given as a decimal value. These results are for networks with both a single convolution as well as networks with two convolutions, and the base optimizer is Adam. All speedups over $3x$ are presented as the same color to preserve detail in the heatmap.

of our network with reasonable stability. In the Figure 4.4 we can see that these parameters indeed yield acceleration in all but one case where PKA never reaches the maximum accuracy of Adam, or the minimum loss of Adam. This is discussed further in the Discussion.

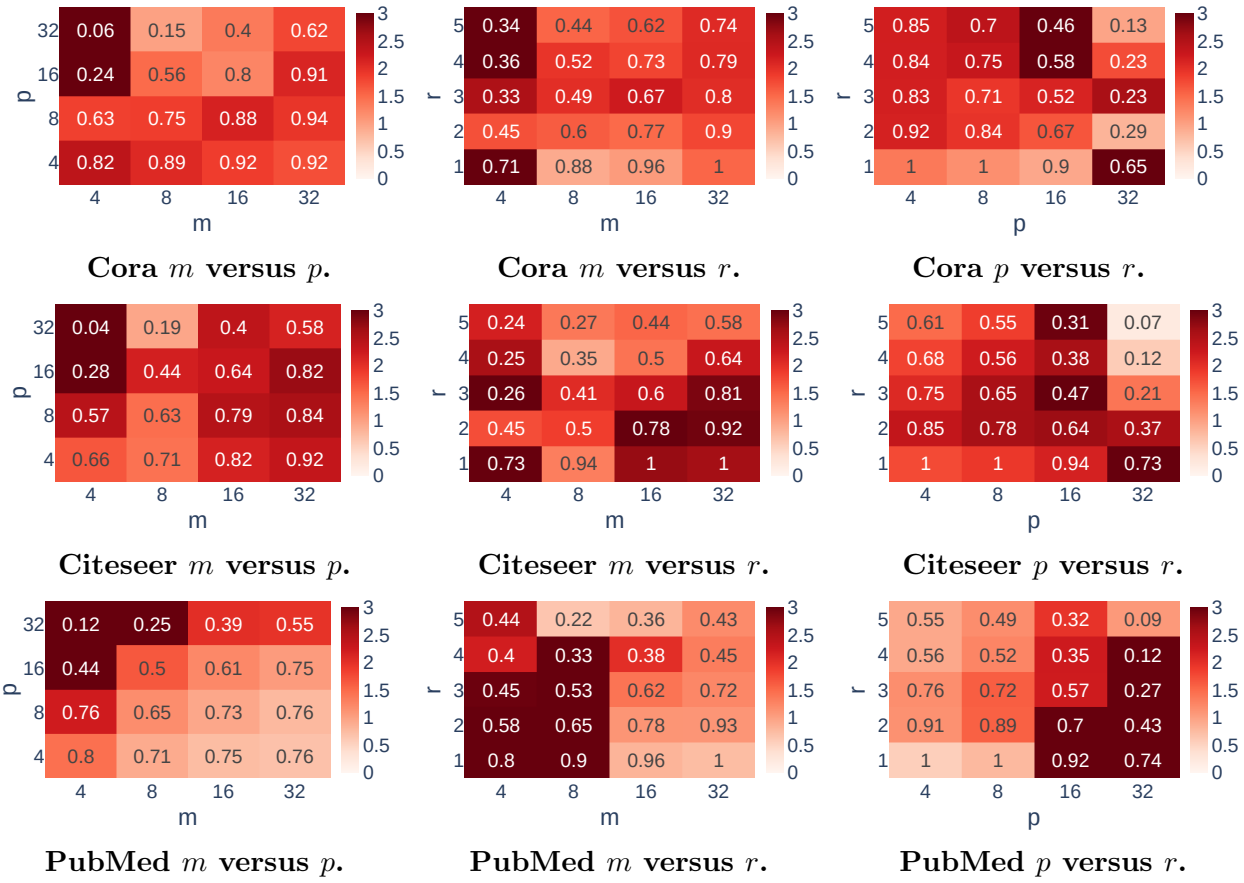
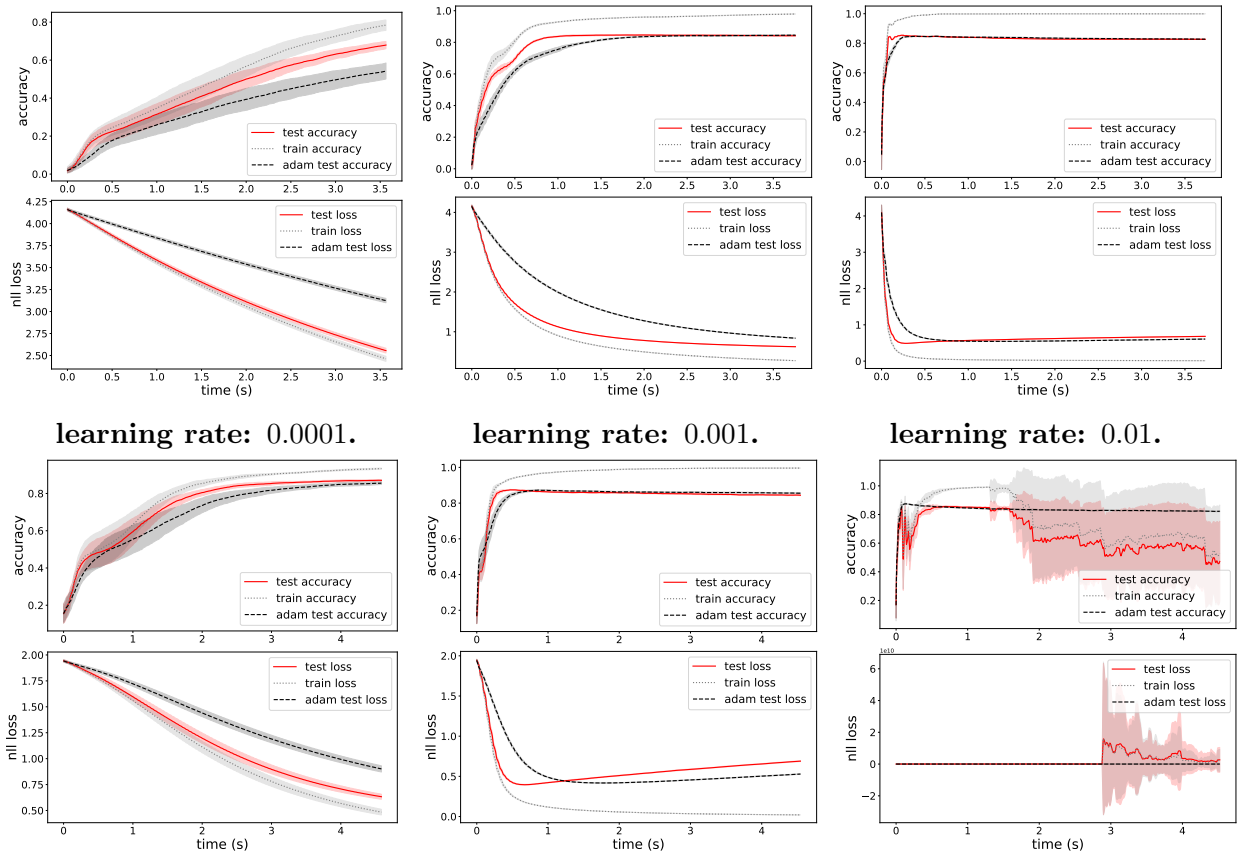


Figure 4.3: Minpoint loss speedup. The average minpoint loss speedup is shown for each pair of parameters (m, p), (m, r), (p, r), and each dataset Cora, Citeseer, and PubMed. Darker coloring implies greater speedup for our method. The proportion of stable runs is given as a decimal value. These results are for networks with both a single convolution as well as networks with two convolutions, and the base optimizer is Adam. All speedups over $3x$ are presented as the same color to preserve detail in the heatmap.

4.6 Discussion

4.6.1 Performance

Our method shows impressive speed-ups when parameters are well chosen. As suggested in Figures 4.2 and 4.3, to minimize instabilities p and r should tend towards the bottom of their ranges, while m should be as high as possible while still maintaining a speed-up. This theoretically makes sense, since this means we are forecasting over small time horizons



learning rate: 0.0001. **learning rate:** 0.001. **learning rate:** 0.01.

learning rate: 0.0001. **learning rate:** 0.001. **(a) learning rate:** 0.01.

Figure 4.4: Cora speedups for parameters interpreted from heatmaps. Plots of Cora training for the parameters $m = 4, p = 4, r = 2$ obtained from the heatmaps in Figure 4.2 and Figure 4.3. Three learning rates are shown. The networks in the top row have a single convolution, the bottom row networks have two convolutions. One of these runs (4.4a) is unstable and experiences exploding loss, while the other runs all present speedups in both accuracy and loss improvement.

and have many observations. However, one may be able to ignore this prescription and obtain impressive speedups because of it. Since any speedup over three times is represented as the same color in the heatmaps, these results somewhat conceal how fast patchwork Koopman can be in extreme cases. In Table 4.2, we show the mean and max speed-ups for all experiments whose parameter combinations are stable and achieve the minimum loss or maximum accuracy obtained by Adam for the experiment. We additionally remove outliers from this dataset beyond two standard deviations above the mean. We can see that for the proper combination of parameters and problems, patchwork Koopman yields incredible speedups.

Table 4.2: Mean and max speedups for all runs which achieve the desired accuracy or loss. Outliers beyond two standard deviations were removed before computing.

	Cora	Citeseer	PubMed
Loss speedup mean	2.233	3.323	2.517
Loss speedup max	11.483	18.5	23.233
Accuracy speedup mean	1.615	2.689	1.851
Accuracy speedup max	4.078	10.488	8.256

Beyond speedups, our method requires minimal memory. Any Koopman training method will require a minimum of $O(m\omega)$ floats for tracking memory and $O((m-1)\omega + 2(m-1)^2)$ for computing the SVD. In [68], they require a very large m. This is because they only use the Koopman approximation for the final steps of training near convergence, and they track the state for a long time horizon to ensure accuracy. Large m values quickly become unfeasible on GPU. Alternatively, we use a method similar to [69] which alternates between Koopman training and standard training techniques. This requires a much smaller m value that is easier to store on GPU.

4.6.2 Best practices for mitigating instabilities

As can be seen in Figure 4.4, cases may arise when the parameter choice for patchwork Koopman yields exploding loss and accuracy diminishes. The first reason this arises is numerical stability. Because we are iterating matrices on vectors in Algorithm 6, we can loosely analyze this method in terms of power iteration. As p increases, $U^p v$ approaches the eigenvector associated with $\|U\|_2$ for a random vector v. Because of this, one avenue for improving stability is to reduce p. This means that it is generally best practice to not "over-predict" when using patchwork Koopman. Alternatively, when $\|U\|_2$ is comparatively large, it will require less iterations p to achieve a similar loss increase. This suggests one may reduce $\|U\|_2$ in order to improve stability. In our implementation this is done through the parameter r. Recalling that $U_i = Y_i X_i^\dagger$, we know $\|U_i\|_2 \leq \|Y_i\|_2 \|X_i^\dagger\|_2$, therefore, $\|U_i\|_2$ can be controlled by reducing $\|Y_i\|_2$ or $\|X_i^\dagger\|_2$, respectively. Because r prunes the smallest singular values from X_i, that means it also prunes the largest singular values from $X_i^\dagger$, thus minimizing $\|X_i^\dagger\|_2$. Note this does not guarantee a lack of blow-up, since $\|Y_i\|_2$ may still be

large; however the results in Figures 4.2,4.3 suggest a correlation between a low r value and increased stability. It should additionally be noted that r should not be too low as to not lose out on crucial information for prediction. For our test instances, the suggested minimum r value is 2.

4.7 Conclusions

We presented an algorithm on GPU which uses a Koopman operator approximation to accelerate optimizers for GCN node classification. Our implementation is the first to present on-GPU speed-ups for Koopman training methods over the entire training window. We found that our operator, which we call the "patchwork Koopman operator", can be computed efficiently with the use of popular libraries such as PyTorch and CUDA's gesvdjBatched functionality. We performed a parametric study over the Cora,Citeseer, and PubMed datasets, and we found that our method boasts an average speedup of over three times that of Adam in many cases. It was additionally determined that certain parameter choices may lead to instabilities in the loss function, and we outline general best practices for avoiding these.

4.8 Extensions

Using Koopman operators for training neural networks is an incredibly fresh field of research, with only three publications prior to this work. Because of this there are a wealth of open problems worth exploring. As mentioned in the preceding chapter, patchwork Koopman operators are prone to instabilities when used with adaptive optimizers such as Adam. It is an unanswered question as to whether or not the same may be said for non-adaptive schemes. In Figure 4.5 we show how effective Koopman training can be for SGD.

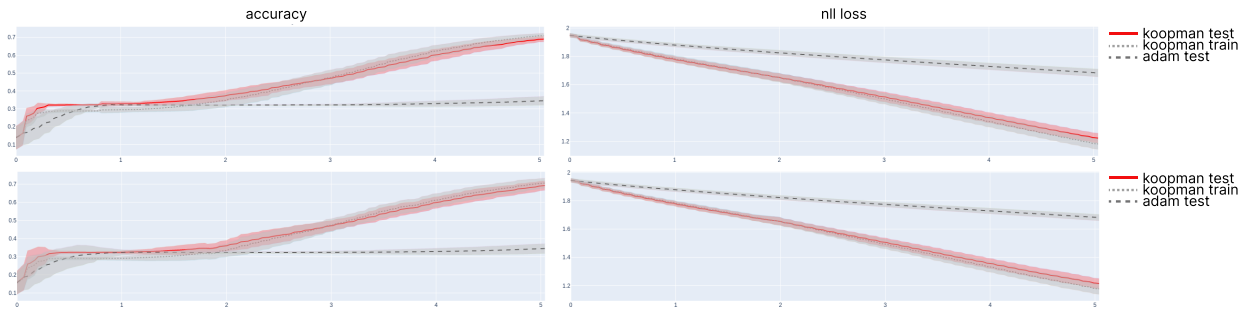


Figure 4.5: SGD Koopman. Results for Koopman accelerated SGD on the Cora dataset on a network with two convolutional layers. In both cases $m = 8$, $p = 16$, and $lr = 0.01$. The top row has $r = 2$, and the bottom row has $r = 3$.

As can be seen in the figure 4.5, Koopman training may provide drastic acceleration for SGD as well as Adam. However SGD may be more stable. Consider the following. Assume during training using Koopman enhanced Adam that the tracked states are all at a high learning rate. Further assume that the learning rate diminishes soon after the final tracked iteration before Koopman approximation. In this case, a purely Adam optimized network may continue to learn, however it is likely that the following Koopman prediction will over-project into the future, increasing loss. In this way non-adaptive schemes may be "safer" with patchwork Koopman optimization since the dynamics are easier to approximate. This suggests one method for improving the stability of patchwork Koopman is to simply change the optimizer we are predicting from. This remains an open question worth investigating.

As previously mentioned, one may be able to obtain more accurate approximations with a "clever" split of the weights. Due to hardware and software constraints we currently can only split the weight vector of our matrix into maximum sizes of 32, and we do this by index. However this is not necesarilly the best course of action. In Figure 4.6 a simple example of our splitting technique is shown on the left. As can be seen each edge in the Koopman operator is independent on the others. This means that the matrix U_i in this case is attempting to predict the next state based on uncorrelated information. Despite this, each red edge on the left hand side of the figure is correlated to edges in the previous layer as shown on the right. A Koopman operator formed using the red edges on the right in the figure 4.6 would likely have an easier time predicting upcoming states.

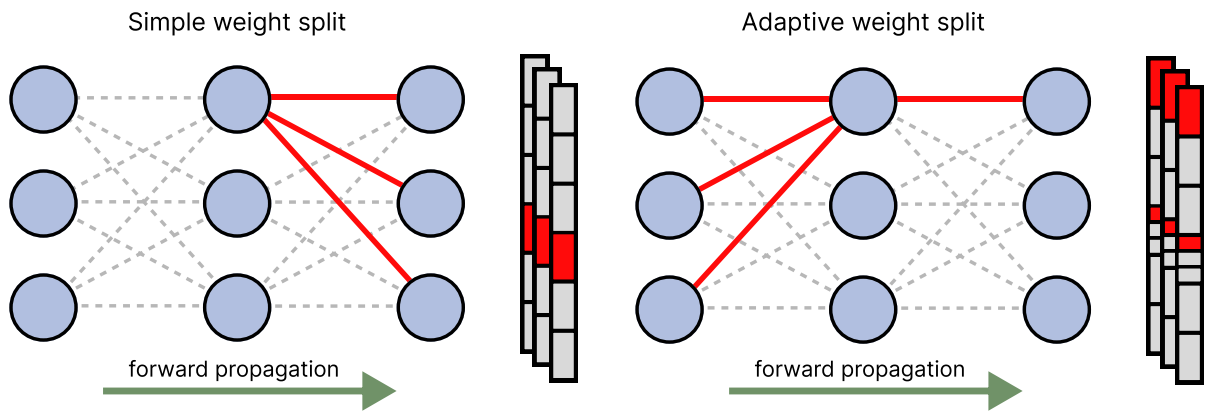


Figure 4.6: Koopman weight splits. Visualization of two different weight splits for patchwork Koopman. Red edges denote edges used to create a sub-Koopman operator.

Such a weight split will likely not be this simple however. For starters, these networks are often deep, and the number of dependencies increases with the number of layers. This poses a significant problem for the imposed maximum of 32 weights per Koopman operator. Instead, one may come up with heuristics for edge-weight correlation within the network, and group them accordingly. Any heuristics which are used must be significantly parallelizable, and have very small computational overhead since our Koopman approximation step must be faster than backpropagation. The best way to approach the subdivision of edges within patchwork Koopman remains an open problem with several avenues for research.

CHAPTER 5
CONCLUSION

In this thesis we explored preprocessing and acceleration techniques for improving the accuracy and timing of node classification and clustering tasks on graphs. New theory was developed for the analysis of null-graph models. This theory was applied in multiple ways to automatically generate new null-graph models with fixed degree sequences in a way which is amenable to parallelization. We also discussed how this may be utilised for an alternative version of modularity maximization, which requires more research. Additionally, we explored the topic of spectral graph coarsening. We discussed how graphs may be coarsened while preserving the Laplcian spectrum and provided a parallel algorithm for greedily performing this task. This represents the first parallel method for explicit spectrum preserving coarsening, which may be useful for multilevel-spectral clustering. We further developed theory relating the difference between the spectrum of the coarsened and original graphs with the per-edge weight approximation which can be recreated from the coarsened graphs weighted topology. This represents the first work to examine this inverse problem. Finally, we investigated how Koopman operators may be used to accelerate GNN node classification. We extended previous CPU only work examining how Koopman theory may be used to accelerate neural networks and implemented Koopman training on GPU. This is the first instance of Koopman training being used for practical speedups throughout the entire neural network training process. Additionally this is the first work to use Koopman training for graph neural networks and node classification. Together, this body of work presents several novel methods which may find applications as important subroutines for many graph data tasks.

REFERENCES

[1] K. Balasubramanian, "Applications of combinatorics and graph theory to spectroscopy and quantum chemistry," *Chem. Rev.*, vol. 85, no. 6, pp. 599–618, Dec. 1985.

[2] J. C. Mitchell, "Social networks," *Annu. Rev. Anthropol.*, vol. 3, no. 1, pp. 279–299, Oct. 1974.

[3] A.-L. Barabâsi, H. Jeong, Z. Néda, E. Ravasz, A. Schubert, and T. Vicsek, "Evolution of the social network of scientific collaborations," *Physica A: Statist. Mech. Appl.*, vol. 311, no. 3-4, pp. 590–614, Aug. 2002.

[4] J. W. Ruge and K. Stüben, "Algebraic multigrid," in *Multigrid Methods*, Philadelphia, PA, USA: SIAM, 1987, pp. 73–130.

[5] J. Kleinberg and E. Tardos, *Algorithm Design*. Noida, India: Pearson Educ. India, 2006.

[6] A.-L. Barabási, "Network science," *Philos. Trans. Royal Society A: Math., Physical, Eng. Sci.*, vol. 371, no. 1987, Mar. 2013, Art. no. 20120375.

[7] J. Gao, B. Barzel, and A.-L. Barabási, "Universal resilience patterns in complex networks," *Nature*, vol. 530, no. 7590, pp. 307–312, May 2016.

[8] C. Jiang, J. Gao, and M. Magdon-Ismail, "True nonlinear dynamics from incomplete networks," in *Proc. AAAI Conf. Artif. Intell.*, 2020, pp. 131–138.

[9] B. K. Fosdick, D. B. Larremore, J. Nishimura, and J. Ugander, "Configuring random graph models with fixed degree sequences," *SIAM Rev.*, vol. 60, no. 2, pp. 315–355, Aug. 2018.

[10] J. Garbus, C. Brissette, and G. M. Slota, "Parallel generation of simple null graph models," in *5th IEEE Workshop Parallel Distrib. Proc. Comput. Soc. Syst. (ParSocial)*, 2020, pp. 1091–1100.

[11] S. Fortunato, "Community detection in graphs," *Physics Rep.*, vol. 486, no. 3-5, pp. 75–174, Feb. 2010.

[12] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, no. 2, Feb. 2004, Art. no. 026113.

[13] F. Chung and L. Lu, "The average distances in random graphs with given expected degrees," *Proc. Nat. Acad. Sci.*, vol. 99, no. 25, pp. 15 879–15 882, Dec. 2002.

[14] M. E. Newman, "Modularity and community structure in networks," *Proc. Nat. Acad. Sci.*, vol. 103, no. 23, pp. 8577–8582, Jun. 2006.

[15] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: Simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, Oct. 2002.

[16] M. Winlaw, H. DeSterck, and G. Sanders, "An in-depth analysis of the chung-lu model," Lawrence Livermore Nat. Lab., Livermore, CA, USA, Tech. Rep. LLNL-TR-678729, 2015.

[17] J. C. Miller and A. Hagberg, "Efficient generation of networks with given expected degrees," in *Int. Workshop Algorithms Models Web-Graph*, 2011, pp. 115–126.

[18] M. Alam, M. Khan, A. Vullikanti, and M. Marathe, "An efficient and scalable algorithmic method for generating large-scale random graphs," in *Proc. Int. Conf. High Perf. Comput., Netw., Storage Anal.*, 2016, pp. 372–383.

[19] G. M. Slota, J. Berry, S. D. Hammond, S. Olivier, C. Phillips, and S. Rajamanickam, "Scalable generation of graphs for benchmarking HPC community-detection algorithms," in *IEEE Int. Conf. High Perf. Comp., Netw., Storage Anal.*, 2019, pp. 1–14.

[20] P. S. Chodrow, "Moments of uniform random multigraphs with fixed degree sequences," *SIAM J. Math. Data Sci.*, vol. 2, no. 4, pp. 1034–1065, Oct. 2020.

[21] T. Britton, M. Deijfen, and A. Martin-Löf, "Generating simple random graphs with prescribed degree distribution," *J. Statis. Physics*, vol. 124, no. 6, pp. 1377–1397, Sep. 2006.

[22] R. van der Hofstad, "Critical behavior in inhomogeneous random graphs," *Random Struct. Algorithms*, vol. 42, no. 4, pp. 480–508, Jul. 2013.

[23] J. J. Pfeiffer III, T. La Fond, S. Moreno, and J. Neville, "Fast generation of large scale social networks with clustering," 2012, *arXiv:1202.4805*.

[24] N. Durak, T. G. Kolda, A. Pinar, and C. Seshadhri, "A scalable null model for directed graphs matching all degree distributions: In, out, and reciprocal," in *IEEE 2nd Netw. Sci. Workshop (NSW)*, 2013, pp. 23–30.

[25] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. C424–C452, Mar. 2014.

[26] G. M. Slota and J. Garbus, "A parallel LFR-like benchmark for evaluating community detection algorithms," in *5th IEEE Workshop Parallel Distrib. Proc. Comput. Soc. Syst. (ParSocial)*, 2020, pp. 1112–1115.

[27] A. Eisinberg, G. Franzé, and P. Pugliese, "Vandermonde matrices on integer nodes," *Numerische Mathematik*, vol. 1, no. 80, pp. 75–85, Jul. 1998.

[28] C. Brissette and G. M. Slota, "Limitations of chung lu random graph generation," in *Int. Conf. Complex Netw. Appl.*, 2021, pp. 451–462.

[29] M. J. Zaki, W. Meira Jr, and W. Meira, *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge, United Kingdom: Cambridge Univ. Press, 2014.

[30] J.-F. Bonnans, J. C. Gilbert, C. Lemaréchal, and C. A. Sagastizábal, *Numerical Optimization: Theoretical and Practical Aspects*. Berlin, Germany: Springer Sci. Bus. Media, 2006.

[31] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos Nat. Lab., Los Alamos, NM, USA, Tech. Rep. LA-UR-08-05495; LA-UR-08-5495, 2008.

[32] G. Wallace, "Gabriel kron, tensor analysis of networks," *Bull. Amer. Math. Soc*, no. 11, pp. 536–538, Jul. 1941.

[33] J. Chen, Y. Saad, and Z. Zhang, "Graph coarsening: From scientific computing to machine learning," 2021, *arXiv:2106.11863*.

[34] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," University of Minnesota, Minneapolis, MN, USA, Tech. Rep. 97-061, 1997.

[35] A. Loukas, "Graph reduction with spectral and cut guarantees.," *J. Mach. Learn. Res.*, vol. 20, no. 116, pp. 1–42, Jan. 2019.

[36] Y. Jin, A. Loukas, and J. JaJa, "Graph coarsening with preserved spectral properties," in *Int. Conf. Artif. Intell. Statist.*, 2020, pp. 4452–4462.

[37] F. R. Chung and F. C. Graham, *Spectral Graph Theory*. Providence, RI, USA: Amer. Math. Soc., 1997.

[38] S. Zelditch, *Eigenfunctions of the Laplacian on a Riemannian Manifold*. vol. 125, Philadelphia, PA, USA: Amer. Math. Soc., Dec. 2017.

[39] M. Kac, "Can one hear the shape of a drum?" *The Amer. Math. Monthly*, vol. 73, no. 4P2, pp. 1–23, Apr. 1966.

[40] C. Gordon, D. L. Webb, and S. Wolpert, "One cannot hear the shape of a drum," *Bulletin Amer. Math. Soc.*, vol. 27, no. 1, pp. 134–138, Jul. 1992.

[41] E. R. Van Dam and W. H. Haemers, "Which graphs are determined by their spectrum?" *Linear Algebra Appl.*, vol. 373, pp. 241–272, Nov. 2003.

[42] E. R. Van Dam and W. H. Haemers, "Developments on spectral characterizations of graphs," *Discrete Math.*, vol. 309, no. 3, pp. 576–586, Feb. 2009.

[43] T. Sahai, A. Speranzon, and A. Banaszuk, "Hearing the clusters of a graph: A distributed algorithm," *Automatica*, vol. 48, no. 1, pp. 15–24, Jan. 2012.

[44] M. H. McIlwain, "Can you hear the size of the vertices? an inverse spectral problem of laplacians on weighted graphs," Ph.D. dissertation, Houston, TX, USA: Rice Univ., 1998.

[45] C. Godsil and G. F. Royle, *Algebraic Graph Theory*. vol. 207, Berlin, Germany: Springer Sci. Bus. Media, 2001.

[46] J. W. Thomas, *Numerical Partial Differential Equations: Conservation Laws and Elliptic Equations*. vol. 33, Berlin, Germany: Springer Sci. Bus. Media, 2013.

[47] D. Ron, I. Safro, and A. Brandt, "Relaxation-based coarsening and multiscale graph organization," *Multiscale Model. Simul.*, vol. 9, no. 1, pp. 407–423, Mar. 2011.

[48] C. Brissette, A. Huang, and G. Slota, "Parallel coarsening of graph data with spectral guarantees," 2022, *arXiv:2204.11757*.

[49] R. Bhatia, *Perturbation Bounds for Matrix Eigenvalues*. Philadelphia, PA, USA: SIAM, 2007.

[50] F. Tung, A. Wong, and D. A. Clausi, "Enabling scalable spectral clustering for image segmentation," *Pattern Recognit.*, vol. 43, no. 12, pp. 4069–4076, Dec. 2010.

[51] S. Zeng, R. Huang, Z. Kang, and N. Sang, "Image segmentation using spectral clustering of gaussian mixture models," *Neurocomputing*, vol. 144, pp. 346–356, Dec. 2014.

[52] H. Jia, S. Ding, X. Xu, and R. Nie, "The latest research progress on spectral clustering," *Neural Comput. Appl.*, vol. 24, no. 7, pp. 1477–1486, Jun. 2014.

[53] H.-T. D. Liu, A. Jacobson, and M. Ovsjanikov, "Spectral coarsening of geometric operators," *ACM Trans. Graph. (TOG)*, vol. 38, no. 4, pp. 1–13, May 2019.

[54] T. Lescoat, H.-T. D. Liu, J.-M. Thiery, A. Jacobson, T. Boubekeur, and M. Ovsjanikov, "Spectral mesh simplification," in *Comp. Graph. Forum*, 2020, pp. 315–324.

[55] H. Chen, H.-T. D. Liu, A. Jacobson, and D. I. Levin, "Chordal decomposition for spectral coarsening," 2020, *arXiv:2009.02294*.

[56] N. Coorporation, "Nvidia tesla v100 gpu architecture," NVIDIA, Santa Clara, CA, USA, Tech. Rep. CAWP-08608, 2017.

[57] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," *ACM Sigplan Notices*, vol. 46, no. 8, pp. 267–276, Feb. 2011.

[58] G. M. Slota, S. Rajamanickam, and K. Madduri, "High-performance graph analytics on manycore processors," in *Int. Par. Dist. Proc. Symp. (IPDPS)*, 2015, pp. 17–27.

[59] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. C401–C423, Sep. 2014.

[60] X. Liu, M. Yan, L. Deng, *et al.*, "Survey on graph neural network acceleration: An algorithmic perspective," Feb. 2022, *arXiv:2202.04822*.

[61] Q. Zhang, M. Zhang, T. Chen, Z. Sun, Y. Ma, and B. Yu, "Recent advances in convolutional neural network acceleration," *Neurocomputing*, vol. 323, pp. 37–51, Jan. 2019.

[62] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, Oct. 2021.

[63] B. O. Koopman, "Hamiltonian systems and transformation in hilbert space," *Proc. Nat. Acad. Sci.*, vol. 17, no. 5, pp. 315–318, May 1931.

[64] S. L. Brunton, M. Budišić, E. Kaiser, and J. N. Kutz, "Modern koopman theory for dynamical systems," 2021, *arXiv:2102.12086*.

[65] J. N. Kutz, S. L. Brunton, B. W. Brunton, and J. L. Proctor, *Dynamic Mode Decomposition: Data-Driven Modeling of Complex Systems*. Philadelphia, PA, USA: SIAM, 2016.

[66] I. Mezić, "Analysis of fluid flows via spectral properties of the koopman operator," *Annu. Rev. Fluid Mech.*, vol. 45, pp. 357–378, Jan. 2013.

[67] P. J. Schmid, "Dynamic mode decomposition of numerical and experimental data," *J. Fluid Mech.*, vol. 656, pp. 5–28, Aug. 2010.

[68] A. S. Dogra and W. Redman, "Optimizing neural networks via koopman operator theory," in *Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 2087–2097.

[69] M. E. Tano, G. D. Portwood, and J. C. Ragusa, "Accelerating training in artificial neural networks with dynamic mode decomposition," 2020, *arXiv:2006.14371*.

[70] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.

[71] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Int. Conf. Mach. Learn.*, 2017, pp. 1263–1272.

[72] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," in *Proc. 32nd Int. Conf. on Neur. Inf. Proc. Sys.*, 2018, pp. 4805–4815.

[73] F. M. Bianchi, D. Grattarola, and C. Alippi, "Spectral clustering with graph neural networks for graph pooling," in *Int. Conf. Mach. Learn.*, 2020, pp. 874–883.

[74] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *Int. Conf. Mach. Learn.*, 2019, pp. 3734–3743.

[75] L. Bottou *et al.*, "Stochastic gradient learning in neural networks," *Proc. Neuro-nımes*, vol. 91, no. 8, p. 12, Nov. 1991.

[76] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv: 1412.6980*.

[77] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics Intell. Lab. Syst.*, vol. 2, no. 1-3, pp. 37–52, Aug. 1987.

[78] J. H. Tu, C. W. Rowley, D. M. Luchtenburg, S. L. Brunton, and J. N. Kutz, "On dynamic mode decomposition: Theory and applications," *J. Comput. Dyn.*, vol. 1, no. 2, pp. 391–421, Sep. 2014.

[79] M. J. Colbrook and A. Townsend, "Rigorous data-driven computation of spectral properties of koopman operators for dynamical systems," 2021, *arXiv:2111.14889*.

[80] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 8026–8037.

[81] R. Nishino and S. H. C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," in *31st Conf. Neural Inf. Process. Syst.*, vol. 6, 2017.

[82] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore, "Automating the construction of internet portals with machine learning," *Inf. Retrieval*, vol. 3, pp. 127–163, Jul. 2000.

[83] K. D. Bollacker, S. Lawrence, and C. L. Giles, "Citeseer: An autonomous web agent for automatic retrieval and identification of interesting publications," in *Proc. 2nd Int. Conf. Auton. Agents*, 1998, pp. 116–123.

[84] Z. Yang, W. Cohen, and R. Salakhudinov, "Revisiting semi-supervised learning with graph embeddings," in *Int. Conf. Mach. Learn.*, 2016, pp. 40–48.

[85] S. Xiao, S. Wang, Y. Dai, and W. Guo, "Graph neural networks in node classification: Survey and evaluation," *Mach. Vision Appl.*, vol. 33, pp. 1–19, Jan. 2022.

[86] Z. Huang, S. Zhang, C. Xi, T. Liu, and M. Zhou, "Scaling up graph neural networks via graph coarsening," in *Proc. 27th ACM SIGKDD Conf. Knowl. Discovery Data Mining*, 2021, pp. 675–684.