Generative graph models subject to global similarity

by

Alana Shine

A Dissertation Presented to the

FACULTY OF THE USC GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

July 2020

Contents

List of Tables

List of Figures

0.1 Abstract

This thesis explores how to design generative graph models around global features that capture the connectivity of graphs. Generative graph models sample from sets of "similar" graphs according to some probability distribution. Random graphs drawn from generative graph models are used across a wide variety of applications for simulation studies, anomaly detection, and characterizing properties of real-world graphs in areas such as social science and network design. For example, in epidemiology random graphs are used as a tool to simulate disease spread which can aid in crafting vaccination strategies (Britton et al., 2007; Fransson and Trapman, 2019). The random graphs model humans and their interactions; these interactions can result in a disease transition (or not) according to a disease spread model. The fact that the random graphs come from a distribution provides a mechanism for researchers to understand if their claims around a disease spread model hold with high probability.

The topology of graphs are critical in determining how random processes, like disease spread, behave. In particular, how connected the graph is measured by a property called *conductance* is especially important for processes like disease spread and information flow (Berger et al., 2005; Ganesh et al., 2005). Conductance measures the smallest ratio of edges leaving to edges contained inside a cut across all cuts in the graph to capture the connectivity of the "bottleneck" cut. Conductance controls how a random process like disease spread behaves because depending on how sparse the connectivity is across the bottleneck cut, the easier it is to contain the disease from spreading from one side to the other.

With so many graph processes we wish to understand (like disease/information spread, network robustness and online behavior) depending on global connectivity measures (like the density of connections between clusters, the presence of sparse cuts and the distribution of shortest-paths between nodes), this work makes a shift toward designing generative graph models that target global connectivity structure over other graph properties. As a motivating example, suppose that a researcher wants to understand how a vaccination protocol performs under a certain disease spread model and knows the typical degree distribution of the social networks he wants to study. To generate random graphs, the researcher uses a generative graph model that samples from the set of all graphs subject to a fixed degree distribution. Unfortunately, random graphs drawn subject to

nearly any degree distribution with high probability have high conductance (Bollobás, 1998). This means that the simulation study will be performed solely on highly connected graphs, omitting any kind of clustering structure that might exist typically in real-world social networks. In order to study the behavior of processes that depend on global connectivity structure, we need alternative models.

A large body of work designs generative graph models using local features, like degrees, or a high-level community structure. This thesis is a departure from this line of study and generates graphs subject to matching global features capable of capturing complex connectivity structure. This thesis presents three new ways to generate graphs using (1) symmetric normalized Laplacian spectra (2) random walks and (3) cut connectivity as features to match. All three of these features capture aspects of the global connectivity structure of the graph. The generative graph models in this thesis sample random graphs that all share global features with a real-world target graph. In order to measure the success of these generative graph models, we look at two objectives: *similarity* and *diversity*. The similarity objective is in reference to the target graph to ensure that the generated graphs are realistic. The diversity objective is to guard against simply copying the target graph or generating duplicates with slight variations. We want to design our models to generate from a large set of graphs with enough variation that they are useful in down-stream tasks. We find our generative graph models are competitive and often perform better in both the similarity and diversity objectives against a variety of benchmarks. We also explore the intrinsic trade-off between the similarity and diversity objectives and how to approach balancing between them when using our generative graph models.

Chapter 1

Introduction

This thesis presents three new generative graph models that each generate random graphs through matching *global* graph properties. Section 1.1 introduces some of the applications of generative graph models. Next, Section 1.2 provides some background on existing generative graph models and the emphasis on matching *local* features. Next, Section 1.3 introduces our generative graph models and explains the features that they match, what makes these features global, and why matching global features is better suited for some tasks than matching local ones. Lastly, we explain two competing objectives for generative graph models (1)*similarity* and (2)*diversity* in Section 1.4,

1.1 Generative graph models have diverse applications

A *generative graph model* is a probability distribution over graphs together with some sort of mechanism to sample graphs from that distribution. They are typically built using real-world graph data or real-world graph properties so that the distribution is supported on graphs that resemble real-world graphs. Real-world datasets that are modeled as graphs include social, biological and computer networks.

One of the key applications of generative graph models is to run simulations of a *contagion* process that takes place on a social network (López-Pintado et al.; Jackson, 2010). Contagion processes begin with an event occurring at some seed nodes followed by the event spreading to other nodes across edges using some sort of probabilistic process. They occur across a wide variety of fields including epidemiology (disease spread), social science (rumor spread), business (cultural fads) and journalism (news spread) to name a few (Britton et al., 2007; Fransson and Trapman, 2019;

Giakkoupis, 2011; Krishnamurthy and Nettasinghe, 2019). A generative graph model provides a graph distribution that allows a researcher to make probabilistic statements about the contagion process. If some contagion behavior happens with high probability on random graphs drawn from a generative graph model and those graphs resemble real-world data, then a researcher can draw some conclusions about how the dynamics will behave in the real-world.

Another application is to understand the effect of fixing certain graph properties by measuring what characteristics are shared among graphs as a result. For example, studying how often a random graph is disconnected by node deletion under various models can help explain what properties are responsible for making a network robust to an attack (Bollobás and Riordan, 2004). Understanding how fixing one graph property affects another can be useful in detecting graph anomalies in the presence of adversarial behavior (Ranshous et al., 2015).

Generative graph models also can lend themselves as reference points when characterizing the strength of a property in a real-world graph data set, such as deciding if clustering coefficients are large or small (Mislove et al., 2007).

1.2 Long line of generative graph models

Given the wide breadth of generative graph model applications, there is a rich body of work around generative graph model design. Most models take one of the following approaches:

1. Choose a graph uniformly at random subject to fixed local properties.

2. Impose a high-level partition.

3. Prescribe a model growing or rewiring process based upon plausible real-world dynamics.

4. Generate graphs from features that are learned from a deep neural network.

We differentiate here between *local* properties and *global* properties. Local properties refer to those that can be computed from sub-graphs (e.g., degrees, triangle count), whereas a global property looks at the whole graph (e.g., clustering, connectivity). We list some of the generative graph models of each type below with a complete discussion in Chapter 4.

1. Local properties fixed for graph generation include the degree distribution (Bender and Canfield, 1978; Molloy and Reed, 1995; Newman et al., 2001), small motifs (Newman, 2009), expected degree distribution (Erdős and Rényi, 1960; Bollobás, 1998; Chung and Lu, 2002; Chung et al., 2003), or joint degree distribution (Orsini et al., 2015; Gjoka et al., 2013).

2. Models imposing high-level partitioning structure primarily are the Stochastic Block model and its variants (Holland et al., 1983; Karrer and Newman, 2011) as well as the more general Kronecker graph model (Leskovec et al., 2010). These models capture some global structure, but have small variability in the treatment of the nodes.

3. Graph growth based models generate graphs with similar structural properties through a consequence of a dynamic process instead of imposing them. They include Preferential Attachment model (Barabási and Albert, 1999), the Forest Fire model (Leskovec et al., 2007), models for web growth (Kumar et al., 2000), and the Watts-Strogatz Small-World model (Watts and Strogatz, 1998).

4. Deep learning provides another alternative to imposing graph features by learning how to build graphs using a dynamic process parameterized by a deep network trained from a graph data set (You et al., 2018). These techniques can work quite well, but differ from the previous methods as they train from a large data set instead of a single graph.

1.3 Moving toward global features

There is a lack of generative graph models built around global graph properties that capture the connectivity structure of a graph. Perhaps the most basic notion of connectivity is the distribution of connectivity across cuts. A distribution over the connectivity across cuts would tell us how many sparse cuts there are. However, to get a full picture about the structure of the graph we also need to understand how and if these cuts partition the graph. Moreover, we could need a lot of cuts.

A more succinct description of graph connectivity is *conductance*. Conductance measures the smallest ratio of edges leaving to the volume of a cut across all cuts in the graph to capture the connectivity of the "bottleneck" cut. It gives us a lower bound on the connectivity across all cuts in the graph. For many reasonable contagion process assumptions, the behavior of the process depends heavily on the conductance (Berger et al., 2005; Ganesh et al., 2005). This means that to characterize how a contagion process behaves on real-world data, simulation studies should be run on graph datasets with realistic conductance.

This move toward global features is motivated by the fact that random graphs subject to most degree distributions with high probability have high conductance (Bollobás, 1998; Chung et al., 2004). Contagion processes spread rapidly on graphs with high conductance. So for real-world graphs with sparse cuts, any simulations studies on graphs drawn subject to a degree distribution are useless. We explore the effect of fixing larger local features like motifs in Chapter 13 and experiments suggest this does little to preserve low conductance cuts. To study how contagion processes behave on graphs with low conductance, we need alternative models.

This thesis builds three different generative graph models from different sets of graph features, all of which are related to conductance: (1) spectra of the symmetric normalized Laplacian (2) random walks and (3) connectivity across graph cuts. Each generative graph model takes a single graph as input and extracts some of its features to match. The distribution over output graphs is designed to bias toward graphs with features "close" to the input's in order to have similar connectivity structure to the input. A more detailed description of our approach and overview of these new generative graph models are discussed in Chapter 5.

Spectral generation generates graphs using the spectra of the symmetric normalized Laplacian which is a matrix representation of the graph (Chapter 7). Cheeger's inequality shows that conductance is characterized by the second smallest eigenvalue of Laplacian (Theorem 1) and more intricate results show that higher-order eigenvalues characterize the connectivity across additional sparse cuts (Lee et al., 2014). The goal of spectral generation is to generate graphs with spectra similar to the input. If the graphs have similar enough spectra, the connectivity across cuts corresponding to their similar eigenvalues will be close.

We build a generative graph model from *random walks* called *Random walk generation* inspired by the NetGAN proposed by Bojchevski et al. (2018). A random walk on a graph is a sequence of nodes where each consecutive node pair is an edge in the graph. NetGAN is a generative graph model built from a neural network that is trained to generate sequences of vertices ("walks") to resemble random walks on the input graph. The frequency that edges appear on these walks are used to construct a probability distribution over graphs. We introduce additional NetGAN training variants, including one that alleviates the risk of disconnected cuts (Chapter 8). During our investigation, we found that the synthetic walks learned by the NetGAN avoid adding edges across cuts that are sparse in the input graph. As a result, cuts in the output graph approximately match the sparsity of the input. Random walk generation builds on this insight and samples graphs directly from random walks on the input graph eliminating the need to train a neural network (Chapter 9). The walks are sampled to discover sparse cuts and aggregated to avoid placing edges across those sparse cuts in the output graph. Sparse cuts are discovered by using the connection between conductance and random walks. The expected number of steps it takes a random walk to leave a cut is inversely proportional to the conductance of the cut. This connection is exploited to discover sparse cuts in Random walk generation.

Our last generative graph model matches the connectivity across cuts directly (Chapter 10). *Cut fix generation* corrects cuts in some naive "baseline" graph to match the connectivity of those cuts in the input graph. The baseline graph can be constructed from some simple features of the input, like number of edges, or through a more extensive algorithm. The cuts are chosen to approximately maximize the difference in connectivity between the baseline graph and the input graph. The cuts that differ the most are corrected first to prevent correcting cuts that nearly match the connectivity in the input and hopefully lessen the number of cuts that need to be corrected for the corrected baseline graph to resemble the input.

1.4 Measuring success: similarity and diversity

Central to the success of generative graph models is that the generated graphs resemble those they are attempting to model. We call this our similarity objective. We measure the similarity objective with respect to the input graph, as the input graph is a real-world example of the graphs we are attempting to model. We compute similarity with respect to various graph features, both those we use to build the generative graph models and other properties of interest such as shortest-path and clustering coefficient. We list all of the graph features we use to compute similarity in Section 3.1.1.

The second objective of generative graph model design is the diversity of the graphs being generated. At the extreme, we can achieve the similarity objective by memorizing the input graph and outputting the input graph every time. However, this type of model can not be used to make

Graph Generator	Graph Feature to Match	Variation Tool
Spectral Generation	Laplacian Spectra	Random Basis
NetGAN (and variants)	Random Walks	Noisy walk distribution (Learned)
Walk Generation	Random Walks	Noisy walk distribution (Edges inserted)
Cut Fix Generation	Cut Connectivity	Subset of cuts matched

Table 1.1: Each generative graph model achieves similarity by matching a graph feature and diversity by applying a variation tool.

general statements about any class of graphs the input graph belongs to. The diversity objective aims to generate from a large set of graphs. Moreover, each graph in the support set should be sampled with non-negligible probability. One way to achieve the diversity objective would be to sample from a large set uniformly. Another would be to sample from a large set with a probability distribution that biases toward graphs based on some sort of similarity importance but still generates all graphs in the set with non-negligible probability. For our models, many include each edge independently with edge specific probabilities. For this type of graph distribution, the *entropy* of the distribution can be computed and used as a measure of diversity.

Details of how we evaluate both the similarity and diversity objectives can be found in Chapter 3. We observe in our experiments that there seems to be a trade off between similarity and diversity: the more probability we place on graphs that closely resemble the input, the smaller the set is of graphs that are generated. We summarize the graph features matched to achieve similarity and the variation tools used to achieve diversity in Table 1.1.

Chapter 2

Notation

2.1 Standard notation

I_n is the $n \times n$ dimensional identity matrix.

0_n is the $n \times n$ dimensional zero matrix.

$\mathbf{0}_n$ is the n dimensional zero vector.

All matrices X are denoted by capital letters with $x_{u,v}$ denoting the u, v-th entry.

We write the transpose of a matrix X as X'.

Vectors $\mathbf{x}$ are bold lower-case unless otherwise indicated with x_i denoting the i-th entry of $\mathbf{x}$.

Sets are curly letters $\mathcal{C}$.

The set $[n]$ is the set of integers from 1 to n.

All probability distributions are discrete with probability mass functions denoted by lower case letters p.

2.2 Graph and accompanying matrix notation

Generative graph models are built from one or more *target* graphs that are passed as *input* to the algorithm that constructs the model. Unless otherwise indicated, the generative graph models we refer to are built from a single target graph $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ with $|\mathcal{V}^*| = n^*$ and $|\mathcal{E}^*| = m^*$. We use both target graph and input graph to refer to $\mathcal{G}^*$.

The *adjacency matrix* A^* of $\mathcal{G}^*$ has $a^*_{u,v} = 1$ if $(u, v) \in \mathcal{E}^*$ and 0 otherwise.

The random graphs drawn from our generative graph models are written as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $|\mathcal{V}| = n$ and $|\mathcal{E}| = m$.

All sets of vertices $\mathcal{V}$ are enumerated from 1 to n so that $\mathcal{V} = [n]$.

The *degree* matrix $D(X)$ of any symmetric matrix X is a diagonal matrix with $d_{v,v} = \sum_{u \in [n]} x_{u,v}$.

The *symmetric normalized Laplacian* matrix $L(X)$ for any symmetric matrix X as $L(X) = I_n - D(X)^{-1/2} X D(X)^{-1/2}$.

All references to *spectra* (unless otherwise indicated) refer to the eigenvalues of $L(X)$: $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_n)$ where $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$. If computing the spectra for some matrix other than $L(X)$, we write $\boldsymbol{\lambda}(X) = (\lambda_1, \lambda_2, \ldots, \lambda_n)$.

Spectral gap and *Fiedler value* both refer to λ_2. The *Fiedler cut* is computed using the eigenvector $\mathbf{x}_2$ corresponding to the Fiedler Value (Definition 5.0.4).

2.3 Graph sampling notation

Many of our generative graph models are built by constructing a symmetric *template matrix* $\tilde{A}$ which is then scaled to a symmetric non-negative *probababilistic adjacency matrix* A with entries $a_{v,u} \in [0, 1]$. Matrix A is called a probababilistic adjacency matrix because each entry can be treated as an edge probability for *randomized rounding* (Definition 2.3.1).

Definition 2.3.1. Randomized Rounding (Symmetric) takes a symmetric matrix A with entries $a_{v,u} \in [0, 1]$ and constructs a new matrix B with entries $b_{v,u} \in \{0, 1\}$. Each entry $b_{v,u} = b_{u,v} = 1$ with probability $a_{v,u}$ and 0 with probability $1 - a_{v,u}$ for $v \leq u$ by independently flipping a coin with success probability $a_{v,u}$.

Randomized rounding on A is equivalent to including edge (v, u) in $\mathcal{E}$ with probability $a_{u,v}$.

We also refer to a non-negative template matrix as a *frequency matrix*.

2.4 Random walk notation

A *Markov chain* is a random discrete process. At each time step t, the chain has a state $\boldsymbol{w}[t]$ that belongs to some state space $\mathcal{X}$. The value that $\boldsymbol{w}[t]$ takes is drawn from a distribution p that depends on the values that a finite number of k previous states took. A Markov chain is k-th order Markovian if for all $t > k$, $p(\boldsymbol{w}[t] = u | \boldsymbol{w}[t-1], \boldsymbol{w}[t-2], \ldots, \boldsymbol{w}[t-k]) = p(\boldsymbol{w}[t-1], \boldsymbol{w}[t-2], \ldots, \boldsymbol{w}[t-k], \ldots, \boldsymbol{w}[2], \boldsymbol{w}[1])$.

A *random walk* is a special case of a Markov chain that takes place on a graph $\mathcal{G}$ where the values that the states take are vertices $v \in \mathcal{V}$. For all $v \in \mathcal{V}$, $p(\boldsymbol{w}[t] = u | \boldsymbol{w}[t-1] = v)$ is non-zero if and only if $(u, v) \in \mathcal{E}$.

The *standard random walk* is *first-order Markovian* meaning the distribution over the vertex traveled to at time t depends only on the vertex at time $t-1$: $p(\boldsymbol{w}[t] = u | \boldsymbol{w}[t-1]) = p(\boldsymbol{w}[t-1], \boldsymbol{w}[t-2], \ldots, \boldsymbol{w}[2], \boldsymbol{w}[1])$. The standard random walk distribution over vertices at time t is uniform over all neighbors of the vertex at time $t-1$.

For first order random walks, we can write the transitions $p(\boldsymbol{w}[t] = u | \boldsymbol{w}[t-1] = v)$ as a *transition matrix* R so that $r_{v,u} = p(\boldsymbol{w}[t] = u | \boldsymbol{w}[t-1] = v)$ and the entries of each row all sum to 1.

A *stationary distribution* over values a Markov state can take is a vector $\boldsymbol{\pi}$ such that $\boldsymbol{\pi} R = \boldsymbol{\pi}$ with π_v denoting the probability $\boldsymbol{w}[t] = v$ as t grows to infinity. Because $\boldsymbol{\pi}$ is a distribution, $\pi_v \geq 0$ for all $v \in \mathcal{V}$ and $\sum_{v \mathcal{V}} \pi_v = 1$.

2.5 Metric notation

We use several metrics for evaluating both the diversity of the output graphs $\mathcal{G}$ and their similarity to $\mathcal{G}^*$ (Chapter 3).

2.5.1 Distribution metrics for similarity

We evaluate our generative graph models based on how well their spectra match, and to emphasize the difference in eigenvalues corresponding to sparse cuts we introduce a semimetric that compares two sorted vectors of the same length that weights differences in the smaller values higher than differences in larger values: $\ell_2^{\mathrm{LW}}(\lambda, \lambda') = \sum_{i=1}^{n} \frac{1}{i}(\lambda_i - \lambda'_i)^2$.

We use two metrics over discrete distributions p and q: *Earth-Mover's Distance* $d_{\mathrm{EMD}}(p, q)$ and *Total-Variation Distance* $d_{\mathrm{TV}}(p, q)$.

Earth-Mover's Distance between p and q computes a *transport plan* f from p to q with respect to a matrix d over the support $\mathcal{X}$ of p and q. Quantity $f_{i,j}$ is the amount of probability mass shifted from $p(i)$ to $q(j)$ so that (1) $p(i) = \sum_{j \in \mathcal{X}} f_{i,j}$ for all i and (2) $q(j) = \sum_{i \in \mathcal{X}} f_{i,j}$ for all j. The first constraint is to ensure all mass is transported out of all $p(i)$ and the second is to ensure all mass is transported into all $q(j)$. The sum $c(f, d) = \sum_{(i,j) \in \mathcal{X} \times \mathcal{X}} d_{i,j} f_{i,j}$ is the amount of "earth" moved weighted by how far it moved. The Earth Mover's Distance is $c(f^*, d)$ where f^* is the transport plan that minimizes $c(f, d)$. In general, computing the Earth-Mover's distance is intractable. When $\mathcal{X}$ is one-dimensional, we can compute it using the following dynamic program. The minimum cost transport plan will move earth from values in $\mathcal{X}$ close to each other in absolute difference, to solve the dynamic program we use $\mathbf{s}$ which is the sorted values of $\mathcal{X}$:

- $f_{1,0} = 0$, initial transportation cost is zero.
- $f_{i+1,i} = p(i) + f_{i+1,i} - q(i)$, for $i = 1, 2, \ldots, n-1$, compute transported mass from $i-1$ to i where a negative values indicates there is not enough mass to fill the current index i and mass needs to be moved in and a positive value means we have too much mass at the current index i and need to transport mass out.
- $d(s_{i+1}, s_i) = s_{i+1} - s_i$, for $i = 1, 2, \ldots, n-1$, the distance from s_{i+1} to s_i is the difference because the values are sorted.
- $y_{i+1,i} = f_{i+1,i} d(s_{i+1}, s_i)$, weight the transported mass by the distance.
- $d_{\text{EMD}}(p, q) = \sum_{i=1}^{n-1} |y_{i+1,i}|$, take the absolute value. Cost will be negative/positive depending on moving mass to lower/higher indices.

The dynamic program is a linear scan, so the cost of computing the Earth Mover's Distance between two distributions supported on a one dimensional vector is linear plus the cost of sorting the support. We use the Earth Mover's Distance to compare one-dimensional vector valued graph properties for which the absolute difference between values in the support have qualitative meaning. For example, we use the Earth Mover's Distance to compare the histograms of shortest-path distances between nodes because a shortest-path of length 5 is more similar to a shortest-path of length 4 than one of length 1.

The Total variation distance between two distributions p an q does not use a distance over the support, and we use this metric when we do not want to define or have a support space distance. The Total variation distance is between p and q is computed as $d_{\text{TV}}(p, q) = \frac{1}{2} \sum |p(i) - q(i)|$.

2.5.2 Diversity metrics

The entropy of a Bernoulli random variable with probability p is $h(p) = -p \ln(p) - (1-p)\ln(1-p)$. The entropy of a matrix X with entries $x_{u,v} \in [0,1]$ is the average entropy of its entries: $H(X) = \frac{1}{n^2}\sum_{u,v} h(x_{u,v})$ where X is dimension $n \times n$.

Chapter 3

Evaluating generative graph models: Similarity and Diversity

A generative graph model samples random graphs $\mathcal{G}$ from graph set $\mathcal{H}$ according to a distribution p. The distribution is built with an algorithm that has access to an input graph $\mathcal{G}^*$. The utility of the model is judged along two objectives (1) *similarity* and (2) *diversity*.

The similarity objective is to ensure $\mathcal{G}$ with high probability is realistic. Because the algorithm is built from a single real world example $\mathcal{G}^*$, similarity is measured in reference to $\mathcal{G}^*$. To compare how similar $\mathcal{G}$ is to $\mathcal{G}^*$, we compare sets of graph features from $\mathcal{G}$ and $\mathcal{G}^*$. These features can be scalars, like the number of nodes or the number of edges. More complex features can be described by distributions, like the distribution of shortest paths between nodes. We describe the metrics we use to compare feature distributions and the features we use in Section 3.1.

The diversity objective is to ensure that the model did not simply memorize $\mathcal{G}^*$. While a point distribution on $\mathcal{G}^*$ would meet any similarity objective, there is no point for a model at all if it simply outputs the input. The diversity objective can be described using:

- The size of $\mathcal{H}$. If only a few graphs are output, then the model is likely useless for downstream tasks.
- The number of graphs in $\mathcal{H}$ that are seen with probability above some threshold. If $\mathcal{H}$ is large but rarely outputs a graph other than $\mathcal{G}^*$, then the size of $\mathcal{H}$ does little to alleviate the problem of memorizing $\mathcal{G}^*$.
- The variability among graphs in $\mathcal{H}$. For example, if all graphs in $\mathcal{H}$ are only one edge different

than $\mathcal{G}^*$, the model has failed to capture anything interesting about $\mathcal{G}^*$.

We discuss some more formal notions of diversity in Section 3.2. The objectives seem to be inherently at odds with each other: as the definition of what it means for two graphs to be similar to each other gets more specific, there are less satisfying graphs. For the definitions of similarity and diversity that we use and our algorithms, there is no formal trade-off. However, we see experimentally that for models that score higher on our similarity objectives come at a diversity cost.

3.1 Quantifying similarity

To measure the similarity between two graphs we compare the similarity of their graph features. Many of the features we are interested in are computed from a subset of nodes or a subgraph and thus form a distribution. We use the *Earth Mover's Distance* (EMD) between the two distributions (Rubner et al., 2000). In our experiments EMD is a well-suited distance measure between distributions for our purpose because it increases when more probability mass needs to be shifted, or it needs to be shifted larger distances. For example, we want to consider graphs more similar if path distances are mostly off by 1 than when they are mostly off by 5. Earth Mover's Distance on one-dimensional distributions can be computed using a dynamic program which is presented in Section 2.5.1.

For comparing the spectra $\boldsymbol{\lambda}$ and $\boldsymbol{\lambda}$' of graphs $\mathcal{G}$ and $\mathcal{G}$', one option is compute the ℓ_2 norm of $\boldsymbol{\lambda} - \boldsymbol{\lambda}'$. One problem with ℓ_2 is it weights the correspondence between all eigenvalues equally. We know from the Cheeger's inequality (Theorem 1) and its variants that the eigenvalues of smaller indices when comparing Laplacian spectra convey the sparsity of the sparsest cuts, which are the most important to match for many applications (Sections 1.1 and 1.3). Instead, to compare spectra we weight each squared difference inversely proportional to its index using the ℓ_2^{LW} norm presented in Section 2.5.1.

3.1.1 Graph features used to quantify similarity

We use the graph features in Table 3.1 to compute similarity, all of which are commonly used to compare the similarity of graphs (Bojchevski et al., 2018; Leskovec et al., 2010).

To compare spectra, we use the ℓ_2^{LW} metric to weight the difference in eigenvalues inversely proportional to their index to give higher weight to the eigenvalues corresponding to sparse cuts.

Features	Abbreviation	Description
Spectrum	Spec	Eigenvalues of the symmetric normalized Laplacian matrix.
Shortest Path Length Distribution	SP	Distribution of lengths of shortest paths from v to u for all vertex pairs (v, u).
Clustering Coefficient Distribution	CC	Writing $T(v)$ for the number of triangles that v participates in, the clustering coefficient of v is $\frac{T(v)}{d_v(d_v-1)}$, the fraction of pairs of neighbors of v that are neighbors of each other.
Betweenness Centrality Distribution	Btwn	Letting $\sigma_{v,u}$ denote the number of shortest paths between v and u, and $\sigma_{v,u}(s)$ the number of shortest paths from v to u that pass through s, the betweenness centrality of s is $\sum_{v,u\neq s} \frac{\sigma_{v,u}(s)}{\sigma_{v,u}}$.
Degree Distribution	Deg	Frequency for each degree.

Table 3.1: Graph features used to compute graph similarity.

To compare the feature distributions, we use the Earth Mover's Distance. For how to compute both ℓ_2^{LW} and the Earth Mover's Distance, refer to Section 2.5.

3.2 Quantifying diversity

The simplest measure of diversity on a class of graphs is the number of non-isomorphic graphs produced. One way to establish two graphs being non-isomorphic is to compute the spectra. However, two graphs having the same spectrum is insufficient to establish isomorphism. We note that cospectral graphs are believed to be rare so computing the spectra of graphs generated and counting the unique spectra is a fair estimate of the number of graphs generated (Wilson and Zhu, 2008).

An alternative measure of diversity is the *entropy* of the distribution p over the class $\mathcal{H}$ of graphs being sampled from. Let $p(\mathcal{G}_i)$ be the probability mass on $\mathcal{G}_i \in \mathcal{H}$. The entropy of p is $-\sum_i p(\mathcal{G}_i) \log p(\mathcal{G}_i)$ (Shannon, 2001). Entropy can be seen as a stricter notion of diversity because while the class of graphs supported by a distribution might be large, only a few graphs may be sampled with high probability. However, for some generative graph models the distribution p over $\mathcal{H}$ is not made explicit.

For many of the generative graph models we discuss in this thesis, the entropy of p can be computed because graphs are sampled by including each edge independently according to a Bernoulli random variable (Chapter 5). The graphs are sampled from the set of all graphs on n vertices according to a probababilistic adjacency matrix A with entries $a_{u,v} \in [0,1]$ using randomized rounding (Definition 2.3.1). For graphs generated this way, the probability mass on $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is:

$$p(\mathcal{G}) = \prod_{(v,u)\in\mathcal{E}} a_{v,u} \prod_{(v,u)\notin\mathcal{E}} (1 - a_{v,u}).$$

If we take a sum over all $\mathcal{E} \subseteq 2^{\mathcal{V}}$, then we can compute the entropy of p. However, this is expensive when n is large. Instead, as a measure of entropy we compute the average entropy of $a_{v,u}$: $H(A) = \frac{1}{n^2}\sum_{v,u} h(a_{v,u})$ (Section 2.5.2).

Another measure of diversity using matrices (either template matrices $\tilde{A}$ or probabilistic adjacency matrices A[1]) is *edge-overlap* (EO). Edge-overlap computes the fraction of mass that $\tilde{A}$ places on entries $(v,u) \in \mathcal{E}^*$):

$$EO(\tilde{A}, \mathcal{G}^*) = \frac{\sum_{(v,u)\in\mathcal{E}^*} \tilde{a}_{v,u}}{\sum_{(v,u)\in\mathcal{V}^*\times\mathcal{V}^*} \tilde{a}_{v,u}},$$

which provides a measure of how much the model is memorizing $\mathcal{G}^*$ rather than produce diverse graphs.

[1]Template matrices and probabilistic adjacency matrices are defined in Section 2.3

Chapter 4

Existing generative graph models

Generative graph models are built to generate graphs that resemble those in the real world. These real world graphs generally have some set of features in common. Therefore, generative graph models aim to identify or approximate those features from real world graph samples. There are a variety of ways to obtain these realistic features. For some models, the realistic features are of a certain type and extracted from a real world graph (or set of real world graphs). For example, a degree distribution or a high-level clustering. We call these features *prescribed* because the type of feature is specified. Other models generate graphs with shared features as a result of an assumption about how the graphs are built through a *dynamic process*. These models generate graphs using that dynamic process and the dynamics create commonalities among the graphs. Alternatively, the features could be learned through *deep* methods that extract features common to a large data set of graphs.

Not only do generative graph models need a scheme for prescribing, realizing, or learning realistic graph features, they need to provide a means to generate graphs with those features in a way that produces a diverse set. To do so, the generative graph models need some sort of variation mechanism built into the model so that the graphs do not all look the same.[1] For the dynamic process models, the graphs are built through the assumed dynamic process so that randomness is introduced through the parameters of that process. For the feature based models, one method is to randomize the graph features that are not prescribed (non-deep) or not common to all the graphs in the data set (deep). For example, the number of edges might be prescribed but where to place them might be random. Another method is to introduce variation in the graphs by perturbing the

[1]Chapter 3 explains the need for generative graph models to generate both similar and diverse graphs.

observed real-world features slightly, such as edge/node deletion/addition as is done in many of the high-level clustering schemes. Another perturbation method in one of the deep learning models is to stop training early so that real-world features are learned imperfectly and these imperfect features are less restrictive than the real ones.

We characterize a number of generative graph models below according to how they identify features to match and what type of features they are in Table 4.1.

Prescriptive (local features)	Config, ERGM (Section 4.1)
Prescriptive (high level partitioning)	SBM, Kronecker (Section 4.2)
Prescriptive (global features)	**Spectral generation**, **Random walk generation**, **Cut fix generation**, ModulGen (Section 4.3)
Dynamic Process	BA, Small-world (Section 4.4)
Deep (features learned)	Graph VAE, Li et.al. Graph Net, GraphRNN (Section 4.5.2)
Deep/Prescriptive	NetGAN, MolGAN (Section 4.5.3)

Table 4.1: Catalog of generative graph models (models introduced by this thesis are bolded).

4.1 Prescriptive and local

Two general schemes that prescribe local features are *configuration* type models and *exponential random graph* type models.

4.1.1 Configuration model

The original configuration model matches the degree of each node. More generally, configuration models match the number of motifs adjacent to each node. The randomness in the graphs appears from how the nodes interact with each other through these motifs.

The generation of a random graph with the configuration model can be performed through constructing a random bipartite graph. To match the degrees d_v of a input graph $\mathcal{G}^*$, a bipartite graph is constructed with n^* vertices on the left representing nodes and m^* vertices on the right representing edges. Each vertex v on the left has d_v stubs. Each vertex e on the right has 2 stubs so there are exactly $2m^*$ stubs on the left and $2m^*$ stubs on the right. Any random matching of the $2m^*$ vertex stubs to edge stubs corresponds to a graph with edges $e = (v, u)$ where the two stubs incident to vertex e on the right is matched to stubs adjacent to v and u on the left. A self-loop appears if there exists a path $v - e - v$ in the graph, matching a node vertex v to an edge vertex e

twice. Multi-edges appears for any cycles $v - e_1 - u - e_2 - v$. The degree distribution is matched exactly if self-loops and multi-edges are counted as edges.

Karrer and Newman have generalized the configuration model to any motif Newman (2009); Karrer and Newman (2010). For example, under Karrer and Newman the distribution of triangles can be fixed. For each vertex v, include a vertex on the left side of the bipartite graph as before. An edge stub is included for all edges (v, u) that are not part of any triangle. A triangle wedge stub is included for any pairs of edges $((v, u), (u, t))$ for which $(v, s) \in E$ and thus forms a triangle. If triangles adjacent to v do not share any edges, $d_v = s_v + 2 * w_v$ where s_v counts the number of edge stubs and w_v counts the number of triangle wedges adjacent to v. On the right side, a node for every triangle and one node for every edge not part of any triangles is included. Each triangle node is attached to three triangle wedge stubs and each edge node attached to two edge stubs. The graph is then sampled by forming matching between node stubs and motif stubs for both the edge and triangle stub type. An example of a matching between node stubs and motif stubs is provided in Figure 4.1.

This framework can be generalized for any graph motif, introducing different types of nodes on the right for each motif and attaching the corresponding stubs to each vertex. Note that for each motif, there can be different types of topological roles each node can play in a graph and thus these require different types of stubs. For example in Figure 4.1, the Kite motif has two types of stubs.

Defining the topological roles in a motif is difficult to compute as the motifs grow larger. Thus, this technique is only used for small motifs and thus local features. As for the probability of creating motifs unintentionally, Karrer and Newman show that the probability of creating any biconnected subgraph [2] unintentionally vanishes as n grows.

4.1.2 Exponential random graph model

The Exponential Random Graph Model (ERGM) defines a random variable Y to denote an adjacency matrix and samples Y from a distribution that biases towards graphs $G = (V, E)$ with user-prescribed features Frank and Strauss (1986). Formally, $Pr(Y = y) = \frac{1}{Z} exp(\theta^T \phi(y))$ where ϕ is a flexible feature-extractor and θ are the parameters to be learned (usually using Markov Chain Monte Carlo). The features used are usually subgraphs.

[2] A biconnected graph is one in which there are at least two vertex-independent paths between any pair of vertices.

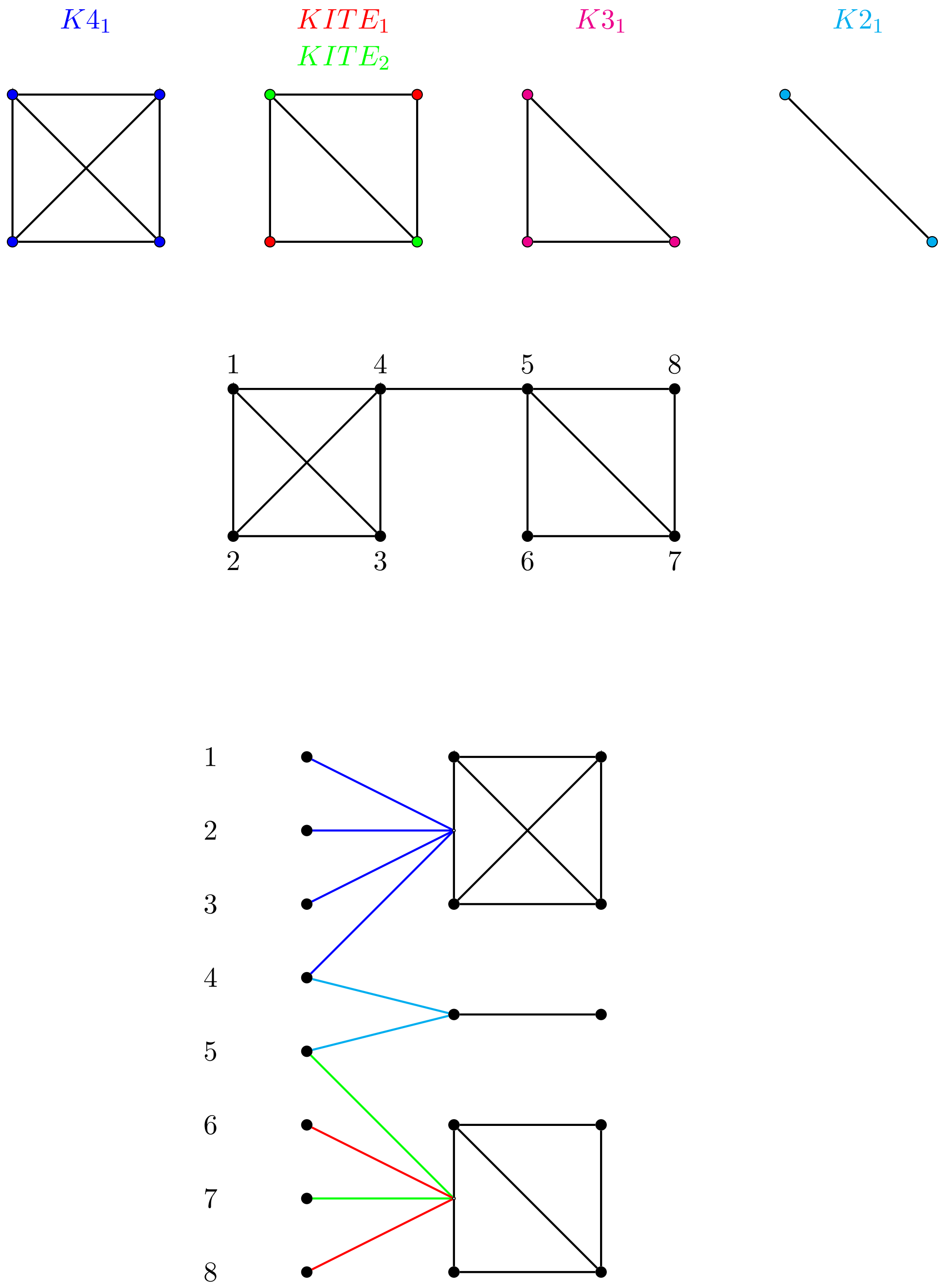


Figure 4.1: Example of configuration model fixing four types of motifs. Each vertex on the left is matched to the motif it participates in. The color of the edge represents the topological role each vertex plays in the motif. Each motif has colored edges corresponding to each topological role it has and how many.

4.2 High-level clustering models

4.2.1 Stochastic Block Models

Stochastic block models (SBM) and following variants bias towards graphs with fixed community structure Holland et al. (1983); Karrer and Newman (2011). In the original SBM, the community structure is inter-/intra-cluster edge density for a user-prescribed set of clusters. Each pair of vertices (i, j) is labeled with a inter-/intra-cluster edge identity corresponding to the clusters that i and j belong to. Graphs are sampled by including an edge for pair (i, j) with the probability prescribed by the inter-/intra-cluster edge identity of (i, j) so that the inter-/intra-cluster edge densities are matched in expectation.

The performance of SBMs heavily relies on fitting the correct clusters and using enough clusters to represent $\mathcal{G}^*$. One common method to partition the graph to form clusters is called *spectral clustering* (Shi and Malik, 2000). Spectral clustering computes node representations in $\mathbb{R}^k$ from the eigenvectors of the symmetric normalized Laplacian corresponding to the k smallest eigenvalues and clusters the node representations using k-means into k clusters. The idea is that by choosing the top k eigenvectors, each eigenvector represents a sparse cut and each node's entry in each eigenvector reveals which side of the cut the node is on. By the orthogonality of the eigenvectors, these sparse cuts have small overlap so the eigencuts can be used to partition the graph into clusters.

4.2.2 Kronecker Models

The stochastic Kronecker graph model is a variant of the SBM where the probability of each edge is again dependent on identity labels on each node Leskovec et al. (2010). The edge probability of an edge between node i and node j is encoded in $K^{[k]} = K \bigotimes K \cdot \bigotimes K$ which is the Kronecker product applied k times to a 2×2 initiator matrix K (the initiator matrix could be different dimensions, but experiments have shown 2 to be sufficient and thus normally 2 is used). Product $K \bigotimes K'$ for 2×2 dimensional K is written as

$$K \bigotimes K' = \begin{bmatrix} K_{11}K' & K_{12}K' \\ K_{21}K' & K22K' \end{bmatrix}$$

Thus, every entry in $K^{[k]}$ will be a product of k terms from initiator matrix K. This induces a hierarchy of clustering probabilities, introducing more intricate structure than the SBM.

Initiator matrix K fully parameterized the Kronecker model. Graphs of size 2^k are generated by using $K^{[k]}$ as a probabilistic adjacency matrix and including each edge (u, v) with probability equal to matrix entry $k^{[k]}_{u,v}$ (randomized rounding, Definition 2.3.1). The Kronecker models are fit by maximizing the likelihood of generating $\mathcal{G}^*$ from $K^{[k]}$ over initiator matrices K. The likelihood is maximized using stochastic gradient descent.

4.3 Prescriptive and global

Most of the generative graph models that prescribe features to match focus on local features or a high-level clustering. One of the reasons to fix local features is that the methods for generation are often simpler to design then for global features. For local features, a method to aggregate the collection of local features can be based on local decisions. For example, in the configuration models, nodes are matched that participate in the same motifs together. The local decisions are not independent because the choices that remain later are effected by those made in the beginning, but as long as self-loops are allowed, whatever choice are made will not prevent matching the remaining local properties. With global features, this seems a lot harder by the fact that the features capture the entire graph. If trying to match a global feature like connectivity across cuts, it can be difficult to make decisions one cut at a time because earlier choices could make it impossible to satisfy another cut in the set in the later. An advantage to the high-level clustering schemes is that the homogeneous node partitions often help with the analysis.

While there is not as large of a literature for matching global features as there is for local features, there are a few existing generative graph models that match global features.

The *spectrum* of a matrix representation of a graph is a global feature that can capture connectivity structure of the graph. One matrix representation that encodes connectivity information in its spectrum is the symmetric normalized Laplacian. One of the models introduced in this thesis, Spectral generation, generates graphs that approximately match Laplacian spectra (Shine and Kempe, 2019). Prior to this work, there were other spectral generative approaches around the Laplacian that we discuss in Section 7.1.

Matrices other than the symmetric normalized Laplacian can be used to construct graphs as well. Baldesi et al. (2018) propose *ModulGen* that generate graphs by matching the spectrum of the *Modularity* matrix Newman (2006). The Modularity matrix B for a graph $\mathcal{G}^*$ with adjacency matrix A^* is $b_{v,u} = a^*_{v,u,-}\frac{d_v d_u}{n^*}$. It weights edges on low degree nodes higher because it indicates a "stronger" connection because these nodes have so few edges. The Modularity matrix is often used to measure the strength of a clustering by adding the Modularity matrix entries for edges between nodes within the same cluster. The idea is that graphs with similar Modularity matrix spectra should have similar Modularity matrices which mean they will behave similarly under different clustering models. Baldesi et al. (2018) use a low-rank approximation of the Modularity matrix, which is then transformed back to an adjacency matrix, noised, scaled, and truncated, to define edge probabilities $p_{v,u}$ and graphs are generated using randomized-rounding (Definition 2.3.1). The rank of the approximation is a user-specified feature, capturing how different the output graphs are likely to be from the input graph.

4.4 Dynamic

Barabasi and Albert design a preferential attachment model designed to produce scale-free graphs Barabási and Albert (1999). The idea is that the graph grows by choosing a node proportional to it's degree to add an additional neighbor too, implementing a "rich get richer" scheme that many find describes the degree distribution of real world networks. Another desired dynamic is one that shrinks the diameter of the graph. The Watts-Strogratz small world model re-wires the edges in a grid to generate graphs with small diameter and local clustering Watts and Strogatz (1998).

4.5 Deep generative graph models

Unlike the non-deep models discussed in sections 4.1, 4.2, 4.3, 4.4 that build a distribution p over graphs from a single graph $\mathcal{G}^*$, the deep models discussed in this section mainly train from a data set of real-world graphs $\mathcal{H}^*$. These models typically assume that $\mathcal{G} \sim p$ are random variables drawn from a random process involving an unobserved continuous random variable z and learn the parameters of that process. Deep generative models are built from datasets in order to extract common features among the graphs in $\mathcal{H}^*$ and then use these features to build a distribution over

$\mathcal{G}$, circumventing the need to prescribe features that $\mathcal{G}$ should have. There are exceptions to deep generative graph models learning from a data set $\mathcal{H}^*$ instead of a single graph $\mathcal{G}^*$, we discuss one called *NetGAN* below in Section 4.5.3 and again in Chapter 8.

There has been a lot of success using deep generative models to generate Euclidean data, like images (Kingma et al., 2014; Goodfellow et al., 2014). These models are generally unsupervised: a data set of positive examples are given and the task is to generate data that resembles the data set. Recently, there has been work that extends the deep generative model techniques for Euclidean datasets to graph datasets. There are several challenges in doing so, two such challenges are (1) how to represent the data (2) how to deal with permutation invariance.

The representation of Euclidean datasets is straightforward as tensors, however, for graphs it is not clear what representation facilitates the best learning. Adjacency matrices are one option, however, these can quickly become intractable as the size of the graph grows. Furthermore, whatever graph representation we use likely needs to have an accompanying metric space to facilitate training. In order to train a deep generative model to generate similar graphs, we need a notion of what similar means that the model can understand. For Euclidean datasets, there are Euclidean metrics that measure how close two data points are. These metrics are often used in the loss function to guide training. However, graph metrics are less straightforward which is further complicated by the fact that in order to incorporate these metrics into the loss they should be differentiable (almost) everywhere.

Permutation invariance is another concern. For most tasks, the quality of a graph is independent of how the nodes are labeled. For these tasks, ideally the loss incurred by the generative model for two isomorphic graphs would be the same. However, for some Euclidean graph representations this seems difficult. For example, for most single layer networks the score assigned to two permuted adjacency matrix inputs would be very different for most weights.

Before discussing a number of prominent deep generative graph models in Sections 4.5.2 and 4.5.3, we define some of the underlying neural net architectures used by the models in Section 4.5.1.

4.5.1 Deep architectures for sampling from distributions

Variational auto-encoder

Variational auto-encoders (VAE) make the assumption that $\mathcal{G} \sim p$ are random variables drawn from a random process involving an unobserved Gaussian variable z (Kingma and Welling, 2013; Doersch, 2016). The goal is to learn the parameters $\boldsymbol{\theta}$ of a feed-forward network $g_{\boldsymbol{\theta}}$ that maps random z to a graph by maximizing the likelihood of graphs $\mathcal{G}^* \in \mathcal{H}^*$:

$$P(\mathcal{G}^*) = \int P(\mathcal{G}^*|z;\boldsymbol{\theta})P(z)dz.$$

However, maximizing this likelihood is intractable over all possible z. For any one X, for most z we will have $P(\mathcal{G}^*|z;\boldsymbol{\theta})$ equal to zero because $g_{\boldsymbol{\theta}}(z) \neq \mathcal{G}^*$. Therefore, we are interested in finding z that *encodes* information about $\mathcal{G}^*$. Building on this intuition, Bayesian tools, and the *Kullback-Liebler* divergence, variational auto-encoders instead maximize the following expectation:

$$\mathbb{E}_{\mathcal{G}^* \sim p^*}[\mathbb{E}_{z \sim Q}[\log P(\mathcal{G}^*|z,\boldsymbol{\theta}) - D_{KL}(Q(z|\mathcal{G}^*,\boldsymbol{\phi})||P(z))]].$$

The expectation is maximized over over parameters (1) $\boldsymbol{\theta}$ and (2) $\boldsymbol{\phi}$ that parameterize (1) a decoder which acts as the generative model and maps hidden variable z to graphs and (2) an encoder which maps graphs to z. The Kullback-Liebler divergence is a probability distribution metric (Kullback, 1997). The terms of the loss can be interpreted as a likelihood term and an error term which measures how well hidden variable z encodes $\mathcal{G}^*$. A VAE is trained using stochastic batch gradient descent by feeding input graphs $\mathcal{G}^* \in \mathcal{H}^*$ to the encoder and then passing the result to the decoder.

Recurrent Neural Network (RNN) and Long-Short Term Memory (LSTM)

A Recurrent Neural Network (RNN) is a chain of feed-forward neural networks connected in a temporal sequence (Bengio, 1993). Each neural network is parameterized by the same vector $\boldsymbol{\theta}$ so that they are all copies of the same network $f_{\boldsymbol{\theta}}$. At each time step t, $f_{\boldsymbol{\theta}}$ is computed from an input X_t and a hidden state h_{t-1} that is computed from the previous round. At time t, $f_{\boldsymbol{\theta}}(X_t, h_{t-1})$ outputs an output o_t and a new hidden state h_t. The idea is that RNNs can be built to capture long-term dependencies to model random processes that have a temporal component (like speech).

A Long-Short Term Memory (LSTM) machine is a special kind of RNN Hochreiter and Schmidhuber (1997) that uses special rules for computing the hidden state s_t. These rules facilitate what should be "memorized" and what should be "forgotten" from past steps in the hidden state.

Generative Adversarial Network (GAN)

The *Generative Adversarial Network (GAN)* is trained using a data set and the model assumes that the data was drawn according to a distribution p^* and the goal of the GAN is to learn that distribution. A GAN consists of two components: (1) the generator: a *generative model* $g_{\boldsymbol{\theta}_g}$ and (2) the discriminator: a *discriminative model* $f_{\boldsymbol{\theta}_d}$ where both are neural networks parameterized by $\boldsymbol{\theta}_g$ and $\boldsymbol{\theta}_d$ which are both vectors (Goodfellow et al., 2014). A generative model is a statistical model of the joint distribution of an observed variable and a target variable. A discriminative model is a statistical model of the conditional probability of the target given the observed variable. The target variable for GANs are binary labels for "real" and "fake". The generator is trying to learn a distribution p^* while the discriminator is trying to tell if data $\boldsymbol{x}$ is drawn from p^* (real) or $g_{\boldsymbol{\theta}_g}$ (fake). The generator and discriminator can be thought of as two players playing a minimax game. The original GAN by Goodfellow et al. (2014) uses the following minimax objective:

$$\min_{\boldsymbol{\theta}_g} \max_{\boldsymbol{\theta}_d} \mathbb{E}_{\boldsymbol{x} \sim p^*}[\log f_{\boldsymbol{\theta}_d}(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x} \sim g_{\boldsymbol{\theta}_g}}[\log(1 - f_{\boldsymbol{\theta}_d}(\boldsymbol{x}))].$$

The objective grows as the discriminator places higher probability on samples drawn from p^* and less probability on samples drawn from $g_{\boldsymbol{\theta}_g}$, so by maximizing the objective the discriminator can differentiate between real and fake samples. Alternatively, the generator wants to "fool" the discriminator and minimize the second term of the objective over $\boldsymbol{\theta}_g$.

Following (Goodfellow et al., 2014), (Arjovsky et al., 2017) introduced another minimax objective called the *Wasserstein* loss. The Wasserstein objective is an approximation of the Earth Mover's

Distance (for definition, see 2.5.1). The *1-Wasserstein distance* between distributions p and q is

$$\sup_{||f||_L \leq 1} \mathbb{E}_{\boldsymbol{x}\sim p}[f(x)] - \mathbb{E}_{\boldsymbol{x}\sim q}[f(x)],$$

where $||f||_L$ is the *Lipschitz-constant* of f is equivalent to the Earth Mover's Distance (Villani, 2008). Therefore, the value of the maximum solution to

$$\sup_{||f_{\boldsymbol{\theta}_d}||_L \leq 1} \mathbb{E}_{\boldsymbol{x}\sim p^*}[f_{\boldsymbol{\theta}_d}(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{x}\sim g_{\boldsymbol{\theta}_g}}[f_{\boldsymbol{\theta}_d}(\boldsymbol{x})]$$

over $\boldsymbol{\theta}_d$ is equivalent to computing the 1-Wasserstein distance. (Arjovsky et al., 2017) shows how minimizing the Wasserstein objective helps prevent the generator from "mode collapse" where the generator outputs only one value. In practice, restricting $f_{\boldsymbol{\theta}_d}$ to 1-Lipschitz functions can only be done approximately so the Wasserstein objective only approximates the Earth Mover's distance. One technique to approximately restrict $f_{\boldsymbol{\theta}_d}$ to 1-Lipschitz functions is a penalty for on the difference between the gradient of $f_{\boldsymbol{\theta}_d}$ and 1 (Gulrajani et al., 2017). Recall that 1-Lipshitz functions should have maximum gradient 1 everywhere and gradient 1 in at least one place. The final training objective then becomes

$$\begin{aligned}\sup_{||f_{\boldsymbol{\theta}_d}||_L \leq 1} \mathbb{E}_{\boldsymbol{x}\sim p^*}[f_{\boldsymbol{\theta}_d}(x)] - \mathbb{E}_{\boldsymbol{x}\sim g_{\boldsymbol{\theta}_g}}[f_{\boldsymbol{\theta}_d}(x)] \\ + \alpha(||\nabla_{\hat{\boldsymbol{x}}} f_{\boldsymbol{\theta}_d}(\hat{\boldsymbol{x}})|| - 1)^2\end{aligned}$$

where $\hat{\boldsymbol{x}}$ is a random interpolation of a $\boldsymbol{x}$ drawn from p^* and another drawn from $g_{\boldsymbol{\theta}_g}$.

4.5.2 Non-prescriptive deep generative graph models

We discuss three prominent deep generative graph networks and how they address both the graph representation and node permutation challenges discussed above.

Graph VAE

GraphVAE learns a mapping from Gaussian random variables z to probabilistic graphs $\mathcal{G}$ using a VAE and trains from a data set $\mathcal{H}^*$ Simonovsky and Komodakis (2018). The graphs are represented as adjacency matrices. To accommodate permutation invariance, GraphVAE adds an additional graph matching term to the loss. During training, on input $\mathcal{G}^*$ and output $\mathcal{G}$, GraphVAE solves a quadratic program that outputs a loss for the number of entries in the adjacency matrices of $\mathcal{G}^*$ and $\mathcal{G}$ that disagree with a permutation that approximately minimizes the loss. The cost of the graph matching along with the size of adjacency matrices prevents Graph VAE from scaling beyond small graphs (size 20).

Graph Neural Network

Li et.al's generative graph model is based on the graph neural network model of Scarselli et.al (Scarselli et al., 2009). Graph neural networks represent graphs by node representations and a sequence of message passing between nodes and their neighbors. In a given message passing round, a node passes its representation to its neighbors. Each node aggregates its own node representation with its neighbors to update its own representation. Subsequent rounds inform each node about larger neighborhoods.

Li uses graph nets to compute the representation of the current graph (Li et al., 2018). Given a graph net computed graph representation, Li.et.al's model learns functions to decide whether to (1) add a node (2) add edges between the new node and nodes in the graph. Once a new node and possible new edges are added, the graph representation is updated using message passing. Thus, the generative graph net learns the dynamics of how a node enters a graph and chooses to attach itself. The ordering of the nodes is very important. Li et.al learns a distribution over graphs and orderings where training minimizes the likelihood using a fixed canonical ordering or uniform orderings.

Graph RNN

Similar to Li et al. (2018), *Graph RNN* models the dynamic process of growing a graph. Instead of using message passing, a RNN at each time step t adds node t to the graph and a $t-1$ length bit vector for the adjacency list of t to the $t-1$ previous vertices (You et al., 2018). More formally, each RNN consists of two networks: a transition network and an output network parameterized by $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$. Transition network $f_{\boldsymbol{\theta}}$ outputs hidden state h_t and the output network $g_{\boldsymbol{\phi}}$ maps h_t to Θ_t where Θ_t specifies the parameters of the distribution over the adjacency list for the node t. For example, Θ_t might be a length $t-1$ length vector of Bernoulli success probabilities. Graph RNN accommodates graphs of arbitrary size because a graph of size n is constructed after n number of steps.

The representation of the graph is a list of adjacency lists where the t-th list is of length $t-1$ and the v-th entry of the list is 1 if there is an edge between node v and node t and 0 otherwise. By side-stepping the need to represent the entire adjacency matrix and instead encoding more "global" properties through the hidden states, Graph RNN is able to scale to large graphs. In order to train Graph RNN, a node ordering must be assigned to the input graphs in order to represent these graphs as adjacency lists of this form and compute the likelihood of the graph. The Graph RNN is trained to maximize the likelihood over all Breadth-First search orderings.

4.5.3 Deep generative graph models that are partially prescriptive

In the previous models, the deep architectures are trained with loss functions that are independent of any specific graph features. This prevents the user from needing to know what graph features are the most useful for a given task. In this section, we explore two deep architectures that are built around specific features in the same vein as the non-deep prescriptive methods. These deep architectures are trained to generate graphs with the prescribed features.

MolGAN

The *MolGAN* architecture uses a GAN and a reward network to generate graphs that resemble real-world molecules De Cao and Kipf (2018). The loss is a combination of the GAN loss, which does not prescribe any features, and the scoring function which penalizes graphs that do not have specified realistic features that appear in molecule graphs. The graphs samples from the generator and the true samples are represented as a vector of node types and 3-dimensional tensor containing an edge type for each edge. With the graph representations being size $O(n^2)$, MolGAN does not scale beyond graphs of small size. Graph samples from the generator and the true samples are passed to the discriminator and the reward network during training.

The discriminator is trained with a classic GAN training loss to differentiate between graphs from the generator and graphs in the true data set (Section 4.5.1). The loss of the generator is computed from both the discriminator and reward network output and minimized so that the discriminator can not differentiate from the true samples (as in classic GAN training) and the graphs receive a high score from the reward network. The reward network is trained to produce scores that match an external trusted software that can recognize molecular graphs. Whatever criteria the software uses to judge a molecular graph from a non-molecular graph is thus integrated into the MolGAN and prescribes graph features that the generated graphs should have. While the GAN alone is trained to find a distribution that generates graphs that resemble the true samples, incorporating the reward network helps guide the generator towards graphs that share features that are used by the software to judge graphs as molecular and these features might be more important then whatever the GAN would discover from discriminator feedback alone.

NetGAN

The NetGAN uses a GAN built from two RNNs to learn a distribution over random walks (Bojchevski et al., 2018). All of the deep generative graph model methods explored thus far have used the deep architectures to sample graph directly. In contrast, NetGAN uses a deep model to sample prescribed features (random walks). Because the GAN learns from a collection of features, NetGAN trains from a single graph $\mathcal{G}^*$ instead of a set of graphs $\mathcal{H}^*$ as the other deep generative graph models do. After the GAN is trained, a collection of synthetic walks are drawn from the generator and used to construct a graph distribution. A more detailed summary of the NetGAN approach is in Section 5.2 with an in-depth discussion in Chapter 8.

Chapter 5

Our approach toward designing generative graph models around global features

This thesis presents three new generative graph models that are built by matching global features:

1. Spectral generation: Generates graphs by matching the spectrum of the symmetric normalized Laplacian (Chapter 7).
2. Random walk generation: Generates graphs by matching random walks (Chapter 9).
3. Cut fix generation: Generates graphs by fixing the connectivity across cuts. Connectivity refers to the number of edges crossing a cut (Chapter 10).

In addition, we provide an in-depth investigation of NetGAN by Bojchevski et al. (2018) with new training variants designed to match the number of connected components (Chapter 8).

Our generative graph models balance between the similarity and diversity objectives by matching these global graph features "well-enough" without restricting the satisfying graph set to only a few graphs (Chapter 3). For all three of the graph features we use for building our models, if we matched them perfectly then the feasible set would be only a few graphs. For matching Laplacian spectra, while there do exist graphs with the same spectrum, such cospectral pairs are rare. For graphs up to size 11 for which enumeration studies are feasible, they show that roughly .2% of graphs on 11 vertices have a cospectral mate (Wilson and Zhu, 2008). With a target graph $\mathcal{G}^*$ rarely having a single cospectral mate, we choose to match the spectra imperfectly. For matching random walks and cuts, we choose to match them imperfectly because identical walk distributions and connectivity across all $2^{\mathcal{V}^*}$ cuts imply isomorphism. In order to match global features imperfectly but well enough

Model	Update Rule	Importance Heuristic	Utility Score
Spectral generation	Deterministic rounding across eigencuts	Eigencuts corresponding to small eigenvalues	Connectivity across eigencuts
NetGAN	Stochastic gradient descent on GAN parameters	Bias walks across sparse cuts	Discriminator's ability to tell real walks from fake
Random walk generation	Add mass for nodes that appear multiple times on same walks	Bias random walks to discover sparse cuts	Cuts discovered by walks
Cut fix generation	Add/remove mass so connectivity across cut matches target	How much cut connectivity differs from truth	Connectivity across cuts

Table 5.1: The update rules, importance heuristics and utility scores across the generative graph models.

to achieve similarity, we construct our graph sampling distribution p through heuristics that choose which part of the global feature sets are most important to match and which we can relax.

More formally, let $\mathcal{G}_n$ be the set of all graphs on $n = n^*$ vertices. One way to construct a distribution p over $\mathcal{G}_n$ is using a probababilistic adjacency matrix A which is a matrix with entries $a_{u,v} \in [0,1]$ through *randomized rounding* (Definition 2.3.1).

To define these edge probabilities, our approach is to iteratively construct A, starting at iteration $i = 0$ and baseline $A(0)$ constructed from $\mathcal{G}^*$. We then update $A(i-1)$ in iteration i to $A(i)$ using some kind of utility function f which maps A and $\mathcal{G}^*$ to an expected similarity score over $\mathcal{G}$ drawn from p_A. Updated matrix $A(i)$ is constructed so that $f(\mathcal{G}^*, A(i)) > f(\mathcal{G}^*, A(i-1))$ to gain progress on the similarity objective. The entropy score of p_A is $H(p_A) = \sum_{(u,v)\in[n]\times[n]} h(a_{u,v})/n^2$ where $h(a_{u,v})$ is the entropy of a random Bernoulli that takes value 1 with probability $a_{u,v}$. The update rules for A, importance heuristics on features, and utility scores for each generative graph model are in Figure 5.1.

5.0.1 Choice of global features

The spectrum of the symmetric normalized Laplacian, random walks, and cut connectivity are all related to a measure of graph connectivity called *conductance*(Definition 5.0.3).

Definition 5.0.1. *Cut connectivity* Cut connectivity is defined with respect to a probababilistic adjacency matrix A of dimension $n^* \times n^*$ with entries $a_{u,v} \in [0,1]$[1]. The connectivity of $\mathcal{S} \subseteq \mathcal{V}^*$

[1]Conductance is defined the same way for symmetric binary matrices.

denoted $\partial(\mathcal{S}, A) = \sum_{u \in \mathcal{S}, v \in \overline{\mathcal{S}}} a_{u,v}$ measures the number of edges crossing the cut with $\overline{\mathcal{S}} = \{v \in \mathcal{V}^* | v \notin \mathcal{S}\}$.

Definition 5.0.2. *Cut conductance* We have graph $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ with adjacency matrix A^*, cut $\mathcal{S} \subseteq \mathcal{V}^*$, and cut connectivity $\partial(\mathcal{S}, A^*)$. The volume $\psi(\mathcal{S}, A^*) = \sum_{u \in \mathcal{S}, v \in \mathcal{V}^*} a^*_{u,v} = \sum_{u \in \mathcal{S}} d_u$ denotes the total edges adjacent to $\mathcal{S}$. The conductance of $\mathcal{S}$ is $\varphi_{\mathcal{G}^*}(\mathcal{S}) = \partial(\mathcal{S}, A^*)/\min(\psi(\mathcal{S}, A^*), \psi(\overline{\mathcal{S}}, A^*))$.

Definition 5.0.3. *Graph conductance* We have graph $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ with adjacency matrix A^*. The conductance of $\mathcal{G}^*$ is $\Phi(\mathcal{G}^*) = \min_{\mathcal{S} \subseteq \mathcal{V}^*} \varphi_{\mathcal{G}^*}(\mathcal{S})$.

Spectra of symmetric normalized Laplacian matrix

The spectra of the symmetric normalized Laplacian are related to the conductance of a graph by Cheeger's inequality and its higher order variations (Theorems 1,3,4).

Theorem 1. *Cheeger's inequality (Chung and Graham, 1997) For a graph $\mathcal{G}^*$ with adjacency matrix A^*, let $L(A^*) = I - D^{-1/2} A^* D^{-1/2}$ denote the symmetric normalized Laplacian of A^*. Let λ_2^* be the second smallest eigenvalue of $L(A^*)$ and $\Phi(\mathcal{G}^*)$ be the conductance of $\mathcal{G}^*$. Then,*

$$2\Phi(\mathcal{G}^*) \geq \lambda_2^* \geq \frac{\Phi(\mathcal{G}^*)}{2}$$

The proof of Cheeger's inequality depends on the connectivity across the *Fiedler cut* defined by eigenvector $\mathbf{x}_2$ that is associated with the *Fiedler value* λ_2. There are multiple ways to define a cut from $\mathbf{x}_2$, one definition of Fiedler cut is in Definition 5.0.4.

Definition 5.0.4. Fiedler cut For symmetric normalized Laplacian L of graph $\mathcal{G}$, let $\mathbf{x} = \mathbf{x}_2$. Let $Q(\mathbf{x}, t)$ be the set of all vertices v with $x_v \leq t$. The Fiedler cut is $Q(\mathbf{x}, x_{v^*})$ where $x_v = \arg\min_{v \in \mathcal{V}} \varphi_{\mathcal{G}}(Q(\mathbf{x}, \mathbf{x}_v))$ and $\varphi_{\mathcal{G}}(S)$ measures the conductance of $S \subseteq \mathcal{V}$ (Definition 5.0.2).

Cut connectivity

Conductance is related to cut connectivity because cut connectivity is part of its definition.

Random walks

The behavior of random walks is closely tied with conductance. For a random walk with transition matrix R and stationary distribution $\boldsymbol{\pi}$, the *mixing time* is the number of steps required to guarantee that wherever the walk is started, with high probability the distribution over states is "close" to $\boldsymbol{\pi}$. The conductance of R is computed a little differently then for a graph, with $\Phi(R) = \min_{\{\mathcal{S}\subseteq\mathcal{V}^* \,:\, \sum_{i\in\mathcal{S}} \pi_i \leq \frac{1}{2}\}} \frac{\sum_{i\in\mathcal{S}, j\in\overline{\mathcal{S}}} r_{i,j}}{\sum_{i\in s} \pi_i}$. Let $r_{i,j}^{(t)}$ denote the i, j-th entry of R^t which denotes the probability of traversing from i to j on the t-th step.

Theorem 2. *Mixing time, Theorem 2.2 (Jerrum and Sinclair, 1989) The relative point wise distance $\Delta(t)$ at time t is*

$$\Delta(t) = \max_{i\in\mathcal{V}^*, j\in\mathcal{V}^*} \frac{|r_{i,j}^{(t)} - \pi_j|}{\pi_j}$$

. With $\pi_{min} = \min_{i\in\mathcal{V}^*} \pi_i$, if $\min_i r_{i,i} \geq \frac{1}{2}$ then

$$\Delta(t) \leq \frac{(1 - \Phi(R)^2)^2}{\pi_{min}}$$

.

Theorem 2 says that the largest deviation in the distribution over states at time t from the stationary distribution is bounded by $\frac{(1-\Phi(R)^2)^2}{\pi_{min}}$. This implies that as the conductance of R is small, then the walks can take a long time to mix. This is because the walks often get stuck on one side of a sparsely connected cut, skewing the distribution toward one side of the cut over the other. In fact, consider the "lazy" random walk that uses the standard random walk with probability $\frac{1}{2}$ and stays at the current node with probability $\frac{1}{2}$. For a walk of this type of length t that starts in a cut $\mathcal{S}$, the probability that the walk stays entirely in $\mathcal{S}$ for all steps is lower bounded by $1 - t\varphi_{\mathcal{G}^*}(\mathcal{S})/2$. For one derivation, see Proposition 2.5 in (Spielman and Teng, 2013)(Proposition 5.0.1).

Proposition 5.0.1. *Time it takes walks to leave cuts, Proposition 2.5 (Spielman and Teng, 2013) The probability a walk with transition matrix $R = (A^*D^{-1} - I_{n^*})\frac{1}{2}$ on $\mathcal{G}^*$ starting in $\mathcal{S} \subseteq \mathcal{V}^*$ of length t stays entirely in $\mathcal{S}$ is at least $1 - t\varphi_{\mathcal{G}^*}(\mathcal{S})/2$.*

5.1 Spectral generation

Spectral generation, aims to sample from the class of all graphs with similar spectra (Shine and Kempe, 2019) in order to match connectivity across a subset of cuts(Chapter 7). In particular, Cheeger's inequality and variants show that approximately matching spectra ensures that we approximately match the connectivity across sparse cuts that avoid overlap (Theorems 1,3,4).

Concretely, we are given a graph $\mathcal{G}^*$ with spectrum $\boldsymbol{\lambda}^* = \lambda_1^*, \lambda_2^*, \ldots, \lambda_n^*$ where λ_i^* are the eigenvalues of $L(A^*)$. We aim to generate a graph $\mathcal{G}$ with spectrum $\boldsymbol{\lambda}$ such that $\boldsymbol{\lambda} \approx \boldsymbol{\lambda}^*$. Central to our approach is characterizing symmetric normalized Laplacian $L = W_{\boldsymbol{\lambda}^*}(X) = X\Lambda^* X'$ where X is an orthonormal eigenbasis and $\Lambda^*$is the diagonal matrix with diagonal entries $\boldsymbol{\lambda}^*$. We are interested in finding X such that $W_{\boldsymbol{\lambda}^*}(X)$ is the symmetric normalized Laplacian of a graph $\mathcal{G}$.

The Spectral generation algorithm can be summarized as:

1. Sample a random orthonormal basis X to construct a candidate Laplacian matrix $W_{\boldsymbol{\lambda}^*}(X)$ with the desired spectrum $\boldsymbol{\lambda}^*$. Use linear programming to find a template matrix $\tilde{A}$ whose Laplacian is approximately $W_{\boldsymbol{\lambda}^*}(X)$. (Section 7.2)

2. Keeping the spectrum fixed, perform a walk on possible orthonormal bases, guided by an objective function that pushes the entries of $\tilde{A}$ close to 0 or 1.(Section 7.3)

3. Use an LP-based algorithm to minimally perturb the entries of $\tilde{A}$ by a matrix E to construct probabilistic adjacency matrix $A = \tilde{A} + E$ with entries inside $[0, 1]$, while maintaining the row sums. This may perturb the spectrum, but the LP's objective is designed to keep the perturbation small.(Section 7.4)

4. Use a combination of deterministic rounding across eigencuts with small eigenvalues and randomized rounding (Definition 2.3.1) of other edges to round A to a graph $\mathcal{G}$.(Section 7.5)

5.2 NetGAN with new variations that target sparse cuts

In recent work, Bojchevski et al. (2018) proposed the NetGAN architecture, which is to our knowledge the first *Generative Adversarial Network* (GAN) architecture for generating graphs resembling a *single* input graph $\mathcal{G}^*$. A GAN is a a neural network that samples from distributions (Section 4.5.1).

The GAN is not used to sample graphs directly, but used to learn a random walk distribution and that random walk distribution is then used to build a generative graph model. At a high level, NetGAN can be described as follows:

1. From the input graph $\mathcal{G}^*$ with n^* nodes and m^* edges, generate a set of random walks.
2. Train a deep GAN to produce similar "random walk" sequences[2].
3. Once the GAN is sufficiently trained according to a *stopping criterion*, use the "walks" it produces to compute a frequency matrix $\tilde{A}$, in which each edge is assigned the frequency with which it occurs in the output random walks.
4. Use an iterative sampling procedure to produce sample graphs $\mathcal{G}$ from $\tilde{A}$, where exactly m^* edges are sampled without replacement, with probabilities proportional to the entries $\tilde{a}_{u,v}$, subject to having at least one edge specifically selected for each node v.

In Bojchevski et al. (2018), it is shown that the graphs $\mathcal{G}$ produced by the NetGAN model perform well on edge prediction with regards to: (1) area under the receiver operating characteristic curve (AUC), and (2) average precision (AP) (Powers, 2011). It is also shown that the graphs perform competitively with other graph generative models with matching several graph features.

Here, our goal is to investigate the NetGAN approach in more detail, with a focus on the role its different components and choices play (Chapter 8). The key point of departure for our analysis is:

Observation 1. *$\mathcal{G}^*$ and the choice of random walk define a density on $\mathcal{E}^*$, and the output of the GAN is projected to an edge frequency matrix $\tilde{A}$. Any ordering in the walks used to train the GAN or ordering on walks output by the GAN are lost in the projection. Thus, the role of the GAN architecture is to noisily map A^* to $\tilde{A}$, preserving desirable structural properties while introducing diversity through randomness. How do the different components contribute to this goal?*

We focus our investigation on the following NetGAN components:

- The random walk distribution. NetGAN generates random walks from $\mathcal{G}^*$ according to some random walk distribution to give to the GAN, Bojchevski et al. (2018) uses the standard

[2]Random walks, as defined in Section 2.4, are defined on graphs with consecutive vertices walking along edges. The GAN generates integers that represent vertices, however, subsequent integers need not represent an edge in any graph.

random walk and a generalization called *node2vec* walk. We introduce training a new random walk, the *Fastest mixing Markov chain* to bias the walks toward edges crossing sparse cuts so that the connectivity across sparse cuts is preserved (Section 8.3).

- The length of random walks used (Section 8.4). As stated in Observation 1, any information encoded about the sequence the nodes appear in the walk is lost by the projection. What is the role of having walks instead of edges?
- The stopping criterion for stopping GAN training early. Bojchevski et al. (2018) stop GAN training before it learns the walk distribution exactly and memorizes A^*. We incorporate global structure directly in to the stopping criterion to help ensure $\tilde{A}$ has seen enough walks to learn the global structure of the graph (Section 8.6).
- Graph sampling methods. Once the GAN is trained, generator random walks are used to construct $\tilde{A}$ which is then used to sample a graph. We introduce an edge-independent scheme that differs from the iterative procedure used by Bojchevski et al. (2018) (Section 8.5).

5.3 Random walk generation

One of the drawbacks of NetGAN is that (1) the map from walks on $\mathcal{G}^*$ to walks from the generator and (2) the map of walks from the generator to $\tilde{A}$ are difficult to understand inherently from the first mapping being built by a neural network. Instead, *Random walk generation* maps directly from walks on $\mathcal{G}^*$ to $\tilde{A}$ (Chapter 9). Instead of introducing edges that do not appear in $\mathcal{G}^*$ through a noisy mapping with a neural network as in NetGAN, Random walk generation expands the set of pairs for which it considers as candidates for edges. Instead of considering only pairs that appear on the walks consecutively as edges, Random walk generation considers all node pairs that appear on the same walk. The idea is that the more often that two nodes appear on the same walk, the more likely that the nodes are connected by a short path in a dense area. By counting node pair frequencies instead of edges, we can hopefully keep dense areas in $\mathcal{G}^*$ dense in $\mathcal{G}$ without memorizing edges to introduce more diversity into the output graphs. Node pair frequencies not only help us capture the dense areas, but sparse cuts as well. If two nodes u and v rarely appear on the same walk, then they are likely separated by a sparse cut so including edge (u, v) with probability proportional to the number of times they appear on the same walk will help keep the cut that separates them sparse.

Random walk generation takes place in rounds. Each round has a cut construction phase and a $\tilde{A}$ update phase and can be summarized as:

1. Generate a batch of walks. The walks are sampled according to two disjoint seed sets so that if the two seed sets are separated by a sparse cut, the walks are unlikely to cross the cut.

2. Increase $\tilde{a}_{u,v}$ proportional to the number of times u and v appeared on the same walk together.

Once all the rounds are complete, scale $\tilde{A}$ to A so that the sum of entries in A add up to m^*. Finally, $\mathcal{G}$ is constructed by performing randomized rounding on A (Definition 2.3.1).

5.4 Cut fix generation

All of the above methods target matching the connectivity across sparse cuts indirectly by matching eigenvalues or random walk behavior, both of which are related to the connectivity across sparse cuts. *Cut fix generation* can start with any naive baseline probababilistic adjacency matrix A and iteratively makes cut corrections on the cuts that differ the most from $\mathcal{G}^*$ (Chapter 10). The Cut fix generation procedure can be summarized as:

1. Cut sampling: Sample a cut $\mathcal{S} \subseteq V^*$ from the set of all cuts that approximately maximizes the difference in edges crossing in A from edges crossing in A^*.

2. Cut correction: Correct $\mathcal{S}$ in A by constructing a perturbation matrix E so that (1) $A' = A + E$ has entries in $[0, 1]$ and (2) the connectivity of $\mathcal{S}$ in A' is the same as in A^*.

3. Repeat with $A = A'$ until a sufficient number of cuts have been corrected according to some stopping criterion.

$\mathcal{G}$ is constructed by performing randomized rounding on A (Definition 2.3.1).

5.5 Evaluation

We use the following benchmark models for comparison to our generative graph models. The implementation details for our all three are in Chapter 12.

Config. The configuration model by Bender and Canfield (1978); Molloy and Reed (1995), generates graphs by matching the degree sequence (Section 4.1.1).

SBM. The Stochastic block model by Holland et al. (1983) matches the inter/intra-cluster edge densities in expectation for a clustering of $\mathcal{G}^*$ (Section 4.2.1). We use *spectral clustering* to cluster the nodes.

Kronecker. The Kronecker model by Leskovec et al. (2010) which imposes a high-level partitioning via a Kronecker product of initialization matrices (Section 4.2.2).

We use a variety of similarity and diversity metrics described in Chapter 3 for comparing the quality of the discrete graphs generated by each model. The results on discrete graphs generated by each model are in Chapter 11.

Chapter 6

Datasets

6.1 Real world datasets

We use a number of real world graphs for $\mathcal{G}^*$, all of which are available for download at https://icon.colorado.edu/#!/networks(Clauset et al., 2016). For all datasets, any directed edges or multi-edges are treated as undirected, simple edges. Any self-loops are removed. If the graph has multiple components, we use the largest component as the target graph.

1. Football, NCAA college football 2000. A network of football games between colleges during a single season.

2. Airport, US airport network (top 500; 2002). A network of flights between 500 commercial airports in the US.

3. Netscience, Scientific collaborations in network science (2006). A network of collaborations between authors of network science papers.

4. Email, Email network (Uni. R-V, Spain, 2003). An email exchange network between members of a university.

5. Health, Adolescent health (1994). A network of friendships obtained through a health survey.

6. Wiki, Wikipedia norms (2015). Links between Wikipedia pages on editorial norms.

7. EuroRoad, Euroroad network (2011). A network of international "E-roads", mostly in Europe.

8. Rome, Rome roads (1999). A network of roads in Rome.

9. Amazon, Amazon pages (2012). A small sample of web pages from Amazon.com and its sister companies.

10. Advogato, Advogato trust network (2009). A network of trust relationships among users on Advogato, an online community of open source software developers.

11. HEPhysics, Scientific collaborations in physics (1995-2005). Collaboration graph for high-energy physicists.

6.2 Synthetic data

To test the ability of the generative graph models to accommodate graphs that do not expand and those with sparse cuts, we also ran experiments using two synthetic graphs.

1. Five Cluster The five-cluster graph is generated using five Erdhos-Renyi $G(n, p)$ graphs on 100 vertices each, with edge density $\frac{1}{2}$. Then, the five clusters are connected in a line with four edges.

2. Grid The grid is two-dimensional with 900 nodes.

6.3 Basic parameters of graphs

The basic parameters of the real world input graphs (of the largest connected component) are summarized in Table 6.1.

	Num Vertices	Num Edges	Spectral Gap	Fiedler Cut Conductance
Football	115	613.0	0.1368	0.1077
Airport	500	2980.0	0.0274	0.0588
Netscience	379	914.0	0.003	0.0048
Five Cluster	500	12381.0	0.0001	0.0001
Email	1133	5451.0	0.1211	0.1493
Health	2539	10455.0	0.0379	0.0512
Wiki	1872	15367.0	0.1295	0.175
Rome	3353	4831.0	0.001	0.011
Amazon	2879	3886.0	0.0686	0.0933
Advogato	5042	40509.0	0.0674	0.0909
Hephysics	11204	117619.0	0.0018	0.0024
Euro road	1039	1305.0	0.0005	0.0063
Grid 900	900	1740.0	0.0029	0.0252

Table 6.1: Basic properties of the input graphs.

Chapter 7

Generating graphs subject to Laplacian spectra

Spectral generation generates a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with graph connectivity similar to $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ by approximately matching the spectrum of the *symmetric normalized Laplacian* matrix so that $\boldsymbol{\lambda}(L(A^*)) \approx \boldsymbol{\lambda}(L(B))$ where the adjacency matrices of $\mathcal{G}^*$ and $\mathcal{G}$ are A^* and B. From now on, we write $\boldsymbol{\lambda}(L(A^*)) = \boldsymbol{\lambda}^*$ and $\boldsymbol{\lambda}(L(B)) = \boldsymbol{\lambda}$. One pivotal result relating graph connectivity to Laplacian spectra is *Cheeger's inequality* (Theorem 1) which relates graph conductance (Definition 5.0.3) to the second smallest eigenvalue λ_2 which we call the *spectral gap*[1]. The result says that a small spectral gap implies small graph conductance, so a small spectral gap implies the existence of at least one sparse cut $\mathcal{S} \subseteq \mathcal{V}^*$. Following Cheeger's inequality, Lee et al. (2014) provide an analog of Cheeger's inequality for higher order eigenvalues and show that for any graph on n vertices, for $k \leq n$ we can find k cuts with conductance characterized by λ_k.

Theorem 3. *(Lee et al., 2014) Theorem 1.1 Let $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ be a d-regular graph with $d_v = d$ for all $v \in \mathcal{V}^*$ and $2 \leq k \leq n$. Quantity $\partial(\mathcal{S}, A^*) = \sum_{v \in \mathcal{S}, u \in \overline{\mathcal{S}}} a^*_{v,u}$ measures the number of edges crossing the cut with $\overline{\mathcal{S}} = \{v \in \mathcal{V}^* | u \notin \mathcal{S}\}$. The volume $\psi(\mathcal{S}, A^*) = \sum_{u \in \mathcal{S}, v \in \mathcal{V}^*} a^*_{u,v}$ denotes the total edges adjacent to $\mathcal{S}$. The conductance of $\mathcal{S}$ is $\varphi_{\mathcal{G}^*}(\mathcal{S}) = \partial(\mathcal{S}, A^*)/\min(\psi(\mathcal{S}, A^*), \psi(\overline{\mathcal{S}}, A^*))$. Let*

$$\Phi_k(\mathcal{G}^*) = \min_{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k} \max_i \varphi_{\mathcal{G}^*}(\mathcal{S}_i),$$

where $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k$ is a partition of $\mathcal{V}^*$. Then,

$$\frac{\lambda_k}{2} \leq \Phi_k(\mathcal{G}^*) \leq O(k^2)\sqrt{\lambda_k}.$$

[1]The second smallest eigenvalue is called the spectral gap because for connected graphs, the smallest eigenvalue of the symmetric normalized Laplacian is zero so the $\boldsymbol{\lambda}$ measures the gap between the smallest eigenvalue and itself.

(Lee et al., 2014) go on to show an tighter upper-bound on the conductance of the k cuts in terms of λ_{2k}.

Theorem 4. *(Lee et al., 2014) Theorem 1.2 Let $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ be a d-regular graph with $d_v = d$ for all $v \in \mathcal{V}^*$ and $2 \leq k \leq n$. Quantity $\partial(\mathcal{S}, A^*) = \sum_{v \in \mathcal{S}, u \in \overline{\mathcal{S}}} a^*_{v,u}$ measures the number of edges crossing the cut with $\overline{\mathcal{S}} = \{v \in \mathcal{V}^* | u \notin \mathcal{S}\}$. The volume $\psi(\mathcal{S}, A^*) = \sum_{u \in \mathcal{S}, v \in \mathcal{V}^*} a^*_{u,v}$ denotes the total edges adjacent to $\mathcal{S}$. The conductance of $\mathcal{S}$ is $\varphi_{\mathcal{G}^*}(\mathcal{S}) = \partial(\mathcal{S}, A^*)/\min(\psi(\mathcal{S}, A^*), \psi(\overline{\mathcal{S}}, A^*))$. Let*

$$\Phi_k(\mathcal{G}^*) = \min_{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k} \max_i \varphi_{\mathcal{G}^*}(\mathcal{S}_i),$$

where $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k$ is a partition of $\mathcal{V}^*$. Then,

$$\Phi_k(\mathcal{G}^*) \leq O(\sqrt{\lambda_{2k} \log k}).$$

Theorems 3 and 4 tells us that if the smallest k eigenvalues of two d-regular graphs are close, then there exists k cuts in both graphs with similar connectivity. While in general we use $\mathcal{G}^*$ with irregular degrees, the idea is that by generating $\mathcal{G}$ with $\boldsymbol{\lambda}(\mathcal{G}) \approx \boldsymbol{\lambda}(\mathcal{G}^*)$, the connectivity of $\mathcal{G}$ should resemble $\mathcal{G}^*$. We use the ℓ_2^{LW} semimetric to measure similarity between spectra because it places an emphasis on the smaller eigenvalues to place an emphasis on matching sparse cuts.

At a high-level, Spectral generation can be described as:

1. Sample a random orthonormal basis X to construct a candidate Laplacian matrix $W_{\boldsymbol{\lambda}^*}(X)$ with the desired spectrum $\boldsymbol{\lambda}^*$. Use linear programming to find a template matrix $\tilde{A}$ whose Laplacian is approximately $W_{\boldsymbol{\lambda}^*}(X)$ (Section 7.2).

2. Keeping the spectrum fixed, perform a walk on possible orthonormal bases, guided by an objective function that pushes the entries of $\tilde{A}$ close to 0 or 1 (Section 7.3).

3. Use an LP-based algorithm to minimally perturb the entries of $\tilde{A}$ by a matrix E to construct probabababilistic adjacency matrix matrix $A = \tilde{A} + E$ with entries inside $[0, 1]$, while maintaining the row sums. This may perturb the spectrum, but the LP's objective is designed to keep the perturbation small (Section 7.4),

4. Use a combination of deterministic rounding across eigencuts with small eigenvalues and randomized rounding (Definition 2.3.1) of other edges to round A to B which defines graph $\mathcal{G}$ (Section 7.5).

Note our goal is to to find $\mathcal{G}$ that approximately matches the spectrum of $\mathcal{G}^*$, not exactly. We relax the problem to matching the spectrum because simulation studies suggest that pairs of graphs that have the same spectrum are rare, let alone a large collection to meet our diversity[2] objective (Wilson and Zhu, 2008).

7.1 Related work

Our goal is directly related to *inverse eigenvalue problems*: constructing matrices with a given spectrum, subject to other structural properties Chu and Golub (2005). Ours is an *inexact* inverse eigenvalue problem, because the spectrum only needs to be matched approximately. While our goal is to capture the structural properties of $\mathcal{G}^*$, our task also includes generating a *diverse* collection of matrices (and corresponding random graphs), making a deterministic construction inadequate given that simulation studies suggest co-spectral graphs are rare (Wilson and Zhu, 2008).

Typical matrices considered in the context of inverse eigenvalue problems include symmetric non-negative and self-adjoint matrices, both of which have known constructions Fiedler (1974); Laffey and Šmigoc (2007); Fickus et al. (2013). Deciding if a spectrum is realized by a real non-negative matrix is NP-hard Borobia and Canogar (2017).

Inverse eigenvalue *graph* problems ask whether a graph exists such that a given associated matrix (e.g., adjacency, random walk, Laplacian) has a given spectrum. Godsil and McKay present a method for generating non-isomorphic co-spectral graphs with respect to the adjacency matrix Godsil and McKay (1982). There has also been work on generating certain families of co-spectral graphs with respect to the Laplacian matrix Merris (1997). We are interested in the (inexact) inverse eigenvalue graph problem with respect to the symmetric normalized Laplacian; with respect to the normalized Laplacian, Butler and Grout Butler and Grout (2011) have shown how to construct some families of (exactly) cospectral graphs with special structure.

Prior to this work, there have been other spectral graph generative approaches White and Wilson

[2]Similarity and diversity objectives described in Chapter 3

(2007); White (2009); Xiao and Hancock (2006). The idea in these approaches is to consider spectral embeddings of the means and covariances of a *set* of graphs, then interpret these embeddings as samples from a distribution, and sample from this distribution, with the goal of interpolating between graphs. The sampling produces a candidate "Laplacian matrix." In general, it is not guaranteed that such a candidate matrix is the Laplacian of any matrix resembling a graph adjacency matrix. In fact, effort here is focused on actually producing a suitable adjacency matrix, and on choosing a "Laplacian matrix" that lends itself to this transformation (Section 7.2). As far as we can tell, Xiao and Hancock (2006) does not describe any such procedure. White and Wilson (2007) suggests thresholding the values of the matrix, including as edges of the graph those pairs (i, j) for which the entries of the Laplacian are most negative. We compare two different thresholding approaches in this vein to our algorithms in Section 7.6, and find that they perform significantly inferior in matching the spectrum of the input graph.

7.2 Relaxed Spectrum Fitting

The goal of relaxed spectrum fitting is to find a matrix L with spectrum $\boldsymbol{\lambda}^*$ such that it is the symmetric normalized Laplacian of some graph. An arbitrary matrix L with spectrum $\boldsymbol{\lambda}^*$ is insufficient, there must exist a corresponding graph that has L as its symmetric normalized Laplacian. Our relaxed spectrum fitting procedure uses the fact that we can write matrices with the desired spectrum $\boldsymbol{\lambda}^*$ as $W_{\boldsymbol{\lambda}^*}(X) = X\Lambda^*X'$ for any *orthonormal basis* X where $\Lambda^*$is the diagonal matrix with $\boldsymbol{\lambda}^*$ on the diagonal. An orthonormal basis is a $n^* \times n^*$ matrix X such that $XX' = I_n^*$. If $W_{\boldsymbol{\lambda}^*}(X)$ is the symmetric normalized Laplacian of a graph, then there exists a symmetric binary matrix B such that $L(B) = I_n^* - D(B)^{-1}BD(B)^{-1} = W_{\boldsymbol{\lambda}^*}(X)$ and $D(B)$ is the diagonal degree matrix of B with row sums on the diagonal so entry $d(B)_{v,v} = \sum_{v,u\in\mathcal{V}^*} b_{v,u}$. Ideally, we would sample from the set of all such orthonormal matrices X such that a corresponding symmetric binary matrix B exists. Unfortunately, we do not know how to do this. Instead, we use linear programming to find a symmetric template matrix $\tilde{A}$ such that $L(\tilde{A}) = W_{\boldsymbol{\lambda}^*}(X)$ where $\tilde{A}$ is (likely) not binary using Linear Programming.

Our linear program builds from an observation about symmetric normalized Laplacians (Lemma 5).

Lemma 5. *The matrix Z is the Laplacian of some symmetric matrix $\tilde{A}$ with positive row sums if* *and only if there exists a vector* $\mathbf{y} > 0$ *(all entries are strictly positive) such that* $Z\mathbf{y} = 0$.

Proof. For the first direction, assume that $Z = L(\tilde{A}) = I - D(\tilde{A})^{-1/2}\tilde{A}D(\tilde{A})^{-1/2}$ for some symmetric matrix $\tilde{A}$ with positive row sums. Let $d_v = \sum_u \tilde{a}_{v,u} > 0$ be the degree of v in $\tilde{A}$, and let $\mathbf{y}$ be the vector with $y_v = d_v^{1/2}$. Clearly, $\mathbf{y} > \mathbf{0}$, and

$$Z\mathbf{y} \;=\; \mathbf{y} - D(\tilde{A})^{-1/2}\tilde{A}D(\tilde{A})^{-1/2}\mathbf{y} \;=\; \mathbf{y} - D(\tilde{A})^{-1/2}\tilde{A}\cdot\mathbf{1} \;=\; \mathbf{y} - D(\tilde{A})^{-1/2}\mathbf{d} \;=\; \mathbf{0}.$$

For the converse direction, let $\mathbf{y} > \mathbf{0}$ be a solution to $Z\mathbf{y} = \mathbf{0}$. Let $Y = \operatorname{diag}(y_1, y_2, \ldots, y_n)$, and define $\tilde{A} = Y^2 - YZY$. The vector of row sums of $\tilde{A}$ is

$$\tilde{A}\cdot\mathbf{1} \;=\; \sum_v y_v^2 - YZY\cdot\mathbf{1} \;=\; \sum_v y_v^2 - YZ\mathbf{y} \;=\; \sum_v y_v^2 \;>\; \mathbf{0}.$$

And because

$$L(\tilde{A}) \;=\; I - Y^{-1}\tilde{A}Y^{-1} \;=\; I - Y^{-1}(Y^2 - YZY)Y^{-1} \;=\; I - I + Z \;=\; Z,$$

Z is indeed the symmetric normalized Laplacian of $\tilde{A}$. $\square$

The vector entries y_v in Lemma 5 are the square roots of the degrees of node v. During the computation, the basis X may be such that the associated matrix $W_{\boldsymbol{\lambda}^*}(X)$ is not the Laplacian of any matrix $\tilde{A}$ with positive row sums.[3] In order to compute an approximate and usable matrix $\tilde{A}$, we relax the constraint that $Z\mathbf{y} = \mathbf{0}$, and instead aim to minimize $||Z\mathbf{y}||_\infty$. We approximate the positivity constraint by requiring that $y_v \geq \epsilon$ for all v, for a very small positive constant $\epsilon > 0$; we then obtain the following linear program:

[3]Because $\lambda_1^* = 0$, the matrix $W_{\boldsymbol{\lambda}^*}(X)$ is always singular. However, the corresponding eigenvector $\mathbf{x_1}$ will typically have negative entries, so we are not guaranteed a vector $\mathbf{y} > \mathbf{0}$.

$$\begin{aligned} &\text{Minimize} && f && && (7.1)\\ &\text{subject to} && \left|\sum_{u=1}^{n^*} z_{v,u} y_u\right| \le f && v = 1, \dots, n^* && (7.2)\\ & && y_u \ge \epsilon && u = 1, \dots, n^*. && (7.3) \end{aligned}$$

The procedure for fitting a graph $\tilde{A}$ is in Algorithm 1

Algorithm 1: SpectralFitting

```
Result: Matrix $\tilde{A}$ with $\lambda(L(\tilde{A})) \approx \boldsymbol{\lambda}^*$
Input: Orthonormal matrix $X$;
Vector of desired eigenvalues $\boldsymbol{\lambda}^*$;
Number of desired edges $m^*$;
Epsilon to impose positive row sums $\epsilon$ ;
$Z = W_{\boldsymbol{\lambda}^*}(X)$ ;
Solve LP (7.1) with $\epsilon$, obtaining solution $\mathbf{y}$ ;
Template matrix $\tilde{A} = Y^2 - YZY$, where $Y = \text{diag}(\mathbf{y})$ ;
Scale $\tilde{A}$ so the sum of entries is equal to $m^*$, $\tilde{A} := \frac{m^*}{\sum \tilde{a}_{v,u}} \cdot \tilde{A}$ ;
```

If $f = 0$, then $L(\tilde{A}) = W_{\boldsymbol{\lambda}^*}(X)$, and we have found a template matrix $\tilde{A}$ such that $L(\tilde{A})$ has exactly the desired spectrum $\boldsymbol{\lambda}^*$. We note that the spectrum is invariant to the scaling in last step in Algorithm 1. We perform the scaling to match the target edge density before perturbing the edges, in order to avoid having to perturb the matrix twice to yield entries in the interval $[0, 1]$. Even in this case, individual entries $\tilde{a}_{v,u}$ of the template matrix may be negative or larger than 1 (and of course fractional): the solution $\mathbf{y}$ only guarantees that the *total* degree of each node is positive.

When $f > 0$, we have that $L(\tilde{A}) \neq W_{\boldsymbol{\lambda}^*}(X)$, meaning that typically $\boldsymbol{\lambda}(\tilde{A}) \neq \boldsymbol{\lambda}^*$ as well, i.e., the spectrum can be perturbed. We show that the deviation in spectrum can be bounded using the LP objective f; the lemma also motivates our choice of LP objective.

Lemma 6. *Let δ be such that $f \le \delta y_i$ for all $i = 1, 2, \dots, n^*$. Then, the perturbed eigenvalues satisfy* $|\lambda_i(L(\tilde{A})) - \lambda_i(Z)| \le \frac{\delta}{1-\delta} \cdot |\lambda_i(Z)|$.

Proof. The degrees under $\tilde{A}$ are $d_v = \sum_u \tilde{a}_{v,u} = y_v^2 - y_v \sum_u z_{v,u} y_u$, so the perturbed Laplacian matrix $L = L(\tilde{A})$ has entries

$$\ell_{v,u} \;=\; \frac{z_{v,u} y_v y_u}{(y_v^2 - y_v \sum_k z_{v,k} y_k)^{1/2} (y_u^2 - y_u \sum_k z_{u,k} y_k)^{1/2}}.$$

Writing $q_v = \frac{y_v}{(y_v^2 - y_v \sum_k z_{v,k} y_k)^{1/2}}$ and $Q = diag(q_1, q_2, \ldots, q_n)$, the perturbed Laplacian satisfies $L = QZQ'$.

Applying Theorem 7 (below), we get that the relative perturbation in eigenvalues is at most

$$|\lambda_i(L(\tilde{A})) - \lambda_i(Z)| \;\leq\; |\lambda_i(Z)| \cdot ||Q'Q - I||_2.$$

The norm of a diagonal matrix is its maximum entry, so

$$||Q'Q - I||_2 \;=\; \max_v \frac{\sum_k z_{v,k} y_k}{y_v - \sum_k z_{v,k} y_k} \;\leq \max_v \frac{f}{y_v - f},$$

by the LP's first constraint. Because $f \leq \delta y_v$ for all v by assumption, we obtain the claimed bound. □

Theorem 7 (Theorem 2.1 of Eisenstat and Ipsen (1995))**.** *Let $\tilde{A} = Q'AQ$, where Q is a nonsingular matrix. Let λ_i and $\tilde{\lambda}_i$ be the eigenvalues of A and $\tilde{A}$, respectively. Then, $|\tilde{\lambda}_i - \lambda_i| \;\leq\; |\lambda_i| \cdot ||Q'Q - I||_2$, for all i.*

7.3 Stiefel Manifold Optimization

While the techniques in Section 7.2 find a good $\tilde{A}$ so $L(\tilde{A}) \approx X\Lambda^* X'$, if X was a bad basis, no $\tilde{A}$ will be satisfactory, and the entries will lie far outside $[0, 1]$ so the Laplacian eigenvalues will be perturbed. To improve the basis before committing to it, we can perform a walk on the Stiefel manifold $\mathcal{S}_n = \{X : X'X = I\}$, using known techniques to locally optimize an objective function that rewards template matrices $\tilde{A}$ with entries close to 0 or 1. Specifically, we define the objective function $F(\tilde{A}) \;=\; \sum_{v<u} (1 - \tilde{a}_{v,u})^2 \tilde{a}_{v,u}^2$, which has minima when all $\tilde{a}_{v,u}$ are in $\{0, 1\}$ and steeply penalizes entries far from 0 and 1.

For any basis $X \in \mathcal{S}_n$, let $\mathbf{y}_X$ be the solution to the linear program (7.1) (applied to $Z = X\Lambda^* X'$), and $Y_X = \mathrm{diag}(\mathbf{y}_X)$. Then, $\tilde{A}(X) = Y_X^2 - Y_X X \Lambda^* X' Y_X$ is the candidate template matrix for X. The objective is then to find a basis $X \in \mathcal{S}_n$ (approximately) minimizing $F(\tilde{A}(X))$.

While the objective function F itself is differentiable (which would allow for a straightforward application of existing manifold optimization techniques), the transformation $X \mapsto \tilde{A}(X)$ is computed via the solution to a linear program. It is not clear how optimal solutions to the linear program (7.1) change with X, and in particular whether $X \mapsto F(\tilde{A}(X))$ is continuous or differentiable. Therefore, few guarantees for known optimization techniques apply in our setting. To compensate, we use a small maximum step size of .0001 in the optimization.

To perform optimization over the Stiefel manifold, we use known iterative optimization techniques. Specifically, we use the *Polar Decomposition retraction scheme* Absil and Malick (2012)[4]. Retraction schemes in each step t identify a search direction η_t in the tangent bundle of the current basis X_t, such that moving in the direction η_t minimizes F. (See Absil et al. (2009) for definitions of the notions of *tangent bundle* and *retraction*.) The scheme moves X_t by a step size τ_t in the direction η_t, then projects it back onto the manifold using a retraction $R_X(\tau)$. A retraction scheme specifies both the retraction $R_X(\tau)$ and search direction η such that $R_X(\tau)$ is a descent direction: the derivative $R'_X(\tau)$ must be equal to the projection of $-\mathrm{grad}(F(\tilde{A}(X_t)))$ onto the tangent bundle at X. The step size τ_t is typically chosen according to the *Armajiro-Wolfe Conditions*, which ensure that at the point $X_t + \tau_t$, the decrease $F(X_t) - F(X_t + \tau_t)$ is proportional to τ (*sufficient-decrease*), but also that τ_t is sufficiently large such that one cannot decrease $F(X_t + \tau_t)$ by taking a larger step (*curvature condition*) Nocedal and Wright (2006). One iteration of the Stiefel manifold optimization can be summarized as follows:

1. Using X_t, solve the template fit LP (7.1) to compute $\tilde{A}(X_t)$.

2. Treat the y_{X_t} computed from LP (7.1) as constant, and using $\mathrm{grad}(F(\tilde{A}(X_t)))$, compute a search direction η_t using a retraction method.

3. Perform a line search technique to find a step size τ_t that obeys the Armajiro-Wolfe conditions Nocedal and Wright (2006).

4. Set $X_{t+1} = X_t + \tau_t \eta_t$.

[4]Manifold optimization techniques can generally be divided into two categories: retraction schemes and geodesic schemes. We use retractions as geodesics are often difficult to compute Absil et al. (2009).

5. Repeat until a local optimum is reached.

We implemented and experimented with two different retraction methods: the Cayley Transform Retraction Wen and Yin (2013) and the Polar-Decomposition Retraction Absil and Malick (2012). Our experiments showed that the Polar-Decomposition Retraction performed better most of the time.

7.4 Template Perturbation

The next step is to compute an additive perturbation matrix E that leaves the row sums (i.e., degrees) of $\tilde{A}$ intact (first constraint of LP (7.4)), and ensures that all entries of $A = \tilde{A} + E$ are in $[0, 1]$ (third constraint). Subject to this, we aim to minimize the weighted sum of absolute perturbations. The variables are $e_{v,u}$ and $q_{v,u} = |e_{v,u}|$ (second constraint) for $v \neq u$ (with $e_{v,v} = 0$ implicitly). Sum $d_v = \sum_u \tilde{a}_{v,u}$ denotes the (fractional) degree of i.

$$\text{Minimize} \qquad \sum_{v \neq u} \frac{q_{v,u}}{d_v^{1/2} d_u^{1/2}} \tag{7.4}$$

$$\text{subject to} \qquad \sum_j e_{v,u} = 0, i = 1, \dots, n \tag{7.5}$$

$$q_{v,u} \geq |e_{v,u}| \text{for all } v \neq u \tag{7.6}$$

$$0 \leq \tilde{a}_{v,u} + e_{v,u} \leq 1 \text{for all } v \neq u \tag{7.7}$$

$$e_{v,u} = e_{u,v} \text{for all } v \neq u \tag{7.8}$$

Keeping the degrees constant ensures that the normalization factors for the Laplacian are the same for $\tilde{A}$ and A. The choice of objective function is justified by the following lemma.

Lemma 8. *Let $\{e_{v,u}\}$ be an optimal solution of the LP (7.4), with objective value f. Then, the eigenvalues of the Laplacians of $\tilde{A}$ and $A = \tilde{A} + E$ are close: $\sum_i |\lambda_i(L(\tilde{A})) - \lambda_i(L(A))| \leq f$.*

Proof. Consider the perturbation $\Delta = L(\tilde{A}) - L(A)$ of the Laplacian matrix. Applying Theorem 9 (below) with $\Phi(x) = |x|$, we obtain that $\sum_i |\lambda_i(L(\tilde{A})) - \lambda_i(L(A))| \leq \sum_i |\lambda_i(\Delta)|$. In the remainder of the proof, we will show that $\sum_i |\lambda_i(\Delta)| \leq f$.

Because Δ is symmetric and real-valued, $\Delta = U^T \Lambda U$ for a diagonal matrix $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$ of eigenvalues λ_i of Δ, and orthonormal U. Let $\hat{\Lambda} = \text{diag}(|\lambda_1|, \ldots, |\lambda_n|)$ be the diagonal matrix of absolute values of Δ's eigenvalues, and $\hat{\Delta} = U^T \hat{\Lambda} U$. Define $\sigma_i = \text{sign}(\lambda_i)$ (with $\text{sign}(0) := 1$), and $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_n)$ to be the diagonal matrix of signs of λ_i, so that $\hat{\Lambda} = \Sigma\Lambda$. Then, $\hat{\Delta} = U^T \Sigma U \Delta = U'\Delta$, where $U' = U^T \Sigma U$ is a unitary matrix. Because U' is unitary, its entries are bounded by 1 in absolute value. We can therefore bound

$$\sum_i |\lambda_i(\Delta)| \;=\; \text{tr}(\hat{\Delta}) \;=\; \text{tr}(U'\Delta) \;=\; \sum_v \sum_u u'_{v,u} \delta_{u,v} \;\leq\; \sum_{v,u} |\delta_{u,v}|.$$

We have shown that the sum of absolute eigenvalues of a (real symmetric) matrix is bounded by the sum of absolute values of its entries[5]. It remains to bound the sum of absolute entries of Δ. Because the degrees (row sums) are the same, i.e., d_v, for $\tilde{A}$ as for A, we obtain that $\delta_{v,u} = \frac{\tilde{a}_{v,u} - a_{v,u}}{d_v^{1/2} d_u^{1/2}}$. Summing their absolute values over all v, u, this is exactly the objective function of the LP (7.4), which we assumed to be bounded by f. □

Theorem 9 (Theorem 1 of Kato (1987))**.** *If A, B are $n \times n$ Hermitian matrices, their eigenvalues can be enumerated in such a way that for every real-valued convex function Φ on $\mathbf{R}$, we have $\sum_i \Phi(\lambda_i(A) - \lambda_i(B)) \leq \sum_i \Phi(\lambda_i(B - A))$.*

7.5 Matrix Rounding

Theorems 3 and 4 indicate that the cuts corresponding to small λ_k are particularly important for global connectivity properties. Therefore, our rounding procedure focuses on matching those eigenvalues first. The main point of comparison is the Fiedler cut $(\mathcal{S}^*, \overline{\mathcal{S}}^*)$ defined by the Fiedler vector $\mathbf{x}_2$ of the input graph $\mathcal{G}^*$; without loss of generality, we assume that $|\mathcal{S}^*| \leq n^*/2$. All eigencuts $\mathcal{S}$ corresponding to eigenvector $\mathbf{x}$ during matrix rounding (including the Fiedler cut) are defined by sign of the entries in $\mathbf{x}$[6] (Definition 7.5.1).

Definition 7.5.1. *Eigencut with signs.* The eigencut $\mathcal{S} \subseteq \mathcal{V}^*$ associated with $\mathbf{x}$ eigenvector of L is the set $\arg\min_{\psi \in \{+,-\}} \{v \;:\; \text{sign}(x_v) = \psi\}$.

[5]We suspect that this must be a well-known fact, but could not find a reference.

[6]Note that this is different from the Fiedler cut definition in Section 2.2. We use the simpler definition of the using signs for the sake of time.

Ideally, we would like to focus on rounding edges across the Fiedler cut of the fractional graph A, so as to match λ_2^*. However, it is possible[7] that the Fiedler cut of A is extremely unbalanced, with one side only having a handful of nodes. Efforts to round such unbalanced cuts are largely misplaced.

Instead, we focus on the first approximately balanced cut defined by an eigenvector $\mathbf{x}_k$ of A. Let $k \geq 2$ be smallest such that the smaller side of the cut defined by (again, w.l.o.g. $\mathcal{S}_k$ rather than its complement) satisfies $|\mathcal{S}_k| \geq \frac{1}{2}|\mathcal{S}^*|$. We define $\mathbf{x} = \mathbf{x}_2$ and $\mathbf{x}' = \mathbf{x}_k$ and refer to $(\mathcal{S}_k, \overline{\mathcal{S}}_k)$ as the *critical cut* (Definition 7.5.2).

Definition 7.5.2. *Critical cut.* Let A be a $n^* \times n^*$ dimensional matrix with entries $a_{u,v} \in [0,1]$. The *critical cut* is a subset of $\mathcal{V}^*$ and computed with respect to $\mathcal{S}^* \subseteq \mathcal{V}^*$ where $\mathcal{S}^*$ is the Fiedler cut of a target graph $\mathcal{G}^*$ computed from signs using the Fiedler vector. Let $\lambda_1, \lambda_2, \ldots, \lambda_n^*$ denote the spectrum of $L(A)$ and $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ be the associated eigenvectors. The critical cut is the eigencut $\mathcal{S}_k$ computed from signs (Definition 7.5.1) associated with $\mathbf{x}_k$ where k is the smallest integer such that $|\mathcal{S}_k| \geq \frac{1}{2}|\mathcal{S}^*|$.

We write $d_v = \sum_u a_{v,u}$. Our heuristic is guided by the standard characterization $\lambda_2 = \mathbf{x}L(A)\mathbf{x}^T = \sum x_v^2 - \sum_{v \neq u} x_v x_u \frac{a_{v,u}}{\sqrt{d_v d_u}}$.

When v and u are on opposite sides of the Fiedler cut,[8] their signs in $\mathbf{x}$ are opposite, making the term $-x_v x_u \frac{a_{v,u}}{\sqrt{d_v d_u}}$ positive in $\mathbf{x}L(A)\mathbf{x}^T$. This motivates scoring each edge (v,u) based on $x_v x_u$ as candidates for removal (rounding $a_{v,u}$ down to 0) or addition (rounding $a_{v,u}$ up to 1) depending on whether or not the λ_k is too large or too small. Experimentally, we find λ_k is too small so we focus on that case. However, the heuristic of rounding edges in such a way suffers from two drawbacks:

1. The degrees d_v and d_u are affected when the edge (v,u) is removed or added; the changes in the terms for edges (v',u) or (v,u') could offset the rounding progress.

2. The Fiedler vector (and other eigenvectors) may change after removing or adding an edge (v,u), making it difficult to compute a set of edges to remove or add one by one.

[7] We found that this only happened on input graphs with spectral gap less than .01 and with sparse edge density (for example, the Euro Road graph).

[8] The intuition behind the reasoning applies to $\mathbf{x}'$ as well, although the variational characterization of λ_k is more complex.

Ideally, after each small change to some $a_{v,u}$, one should recompute the eigenbasis before continuing. This is computationally expensive, so instead we adapt an idea of the *NetMelt* algorithm, which was designed to identify good edge removals/additions to decrease/increase the spectral gap of an *adjacency* matrix Chen et al. (2016). We assign scores for each edge removal and edge addition. When working with the adjacency matrix, each edge removal/addition affects only one entry of the perturbation matrix E. For the Laplacian matrix, an edge removal or addition can change the spectrum also through the changes in the nodes' degrees (and thus the normalization).

To help safeguard against the possibility of perturbing many edges adjacent to the same node, thereby changing the degrees drastically and misestimating additional rounding effects, we designate a *budget* z for how many edges can be removed; once the budget has been exceeded, the Fiedler vector $\mathbf{x}$, eigenvector $\mathbf{x'}$, and scores are recomputed. In addition, if the (fractional) number of edges crossing the Fiedler cut is below 1, we stop the procedure. To keep the overall edge density reasonably constant, we alternate between edge removals (across the cut) and additions (on some side of the cut).

The central part of the rounding algorithm is the following *Critical Cut Rounding* procedure (Algorithm 2). It is called repeatedly from the overall rounding procedure (Algorithm 4), which handles special cases such as "disconnected" probability matrices A and very unbalanced Fiedler Cuts.

$L(A)$ and the eigenvectors are computed only once for each overall iteration of the critical cut rounding procedure. In each iteration, the critical cut is corrected by alternating between rounding entries of A up to 1 (additions) and down to 0 (removals). Each entry is scored by an approximation of how much the addition/removal will change $\lambda(L(A))_k$ and added to the approximate score total r to estimate how close $\lambda(L(A))_k$ is to matching λ_k^* after each alteration. An iteration terminates once the approximate score r reaches $|\lambda_k(L(A)) - \lambda_k^*|/2$, or the algorithm has reached its budget z. At this point, the critical cut and Fiedler cut are recomputed to check whether the algorithm has indeed reached its goal of ensuring $\lambda_k^* \leq \lambda_k \leq \lambda_k^* + c$, and to adapt should the cuts have changed. When the Critical Cut Rounding procedure terminates, either all edges across the critical cut have been considered, or λ_k is a good approximation to the target: $\lambda_k^* \leq \lambda_k \leq \lambda_k^* + c$.

While Critical Cut Rounding is the key component to our rounding approach, we need to take care of two special cases: disconnected template matrices A and extremely unbalanced Fiedler cuts.

Algorithm 2: CriticalCutRound

```
Result: Matrix $A$ with entries $a_{v,u} \in [0,1]$ and $\lambda_k(L(A)) \approx \lambda^*_k$
Input: Matrix $A$ with entries $a_{v,u} \in [0,1]$ ;
Desired spectrum $\lambda^*$ ;
Budget of mass that can be removed from edges crossing cuts $z$;
Closeness threshold $c$ ;
Compute critical cut $\mathcal{S}_k$ associated with eigenvector $\mathbf{x}'$ (Definition 7.5.2) and Fiedler cut $\mathcal{S}^*$ ;
Difference $\delta = |\lambda_k(L(A)) - \lambda^*_k|$ ;
Set flag unchanged flag to False ;
Repeat$|\lambda_k(L(A) - \lambda^*_k| < c$, the sum of entries crossing $\mathcal{S}_k$ or $\mathcal{S}^*$ is less than 1, or unchanged
  is True ;
Compute lists of pairs of nodes removalCandidates and additionCandidates (Algorithm 3). ;
*Round entries $a_{u,v}$ down/up to 0/1 for pairs $(u,v)$ in
  removalCandidates/additionCandidates to minimize $|\lambda_k(L(A)) - \lambda^*_k|$ keeping track of the
  estimated changes of $\lambda_k(L(A))$*;
Initialize $q = 0$, which keeps track of sum of $a_{u,v}$ that has been rounded down across the cut ;
Initialize $r = 0$, which keeps track of the sum of scores of altered pairs $(u,v)$ that estimate
  the change to $\lambda_k(L(A))$ ;
Initialize $y = 0$, which keeps track of the total change in entries in $A$ (where positive and
  negative changes cancel out) ;
while $q < z$ and $r < \delta/2$ and there remains candidates in removalCandidates and
  additionCandidates to be processed do
    if $y \geq 0$ then
        $(u,v)$ is the pair $(u,v)$ in removalCandidates with the largest score. Discard $(u,v)$
          from removalCandidates. Mark $(u,v)$ to be removed.
    else
        $(u,v)$ is the pair $(u,v)$ in additionCandidates with the largest score. Discard $(u,v)$
          from additionCandidates. Mark $(u,v)$ to be added.
    if $r + f(u,v) \leq \delta/2$ then
        Increment $r$ by $f(u,v)$ ;
        if $(u,v)$ to be removed and $(u,v)$ is not the last removal candidate then
            Increment $q$ by $a_{u,v}$ ;
            Decrement $y$ by $a_{u,v}$ ;
            Assign 0 to $a_{u,v}$ ;
        else
            Increment $y$ by $1 - a_{u,v}$ ;
            Assign 1 to $a_{u,v}$ ;
end
Compute critical cut $\mathcal{S}_k$ associated with eigenvector $\mathbf{x}'$ (Definition 7.5.2) ;
If $A$ is unchanged, set unchanged to True ;
```

Algorithm 3: Compute addition/removal candidates

```
Result: Lists of pairs of nodes removalCandidates and additionCandidates ;
Input: Matrix $A$ with entries $a_{v,u} \in [0,1]$ ;
Desired spectrum $\lambda^*$ ;
Initialize lists removalCandidates and additionCandidates to empty lists. Each list will
  contain pairs $(v,u)$ will be sorted from high to low according to a score $f(v,u)$. ;
Assign scores to each pair associated with an estimate of how much rounding up to
  1/rounding down to 0 will change $\lambda_k(L(A))$;
for $(v,u) \in \mathcal{V}^* \times \mathcal{V}^*$ do
    if $x'_v x'_u < 0$ then
        Score $f(v,u) = \frac{-x'_v x'_u a_{v,u}}{\sqrt{d_v d_u}}$. Add $(u,v)$ to removalCandidates.
    else
        Score $f(v,u) = \frac{x'_v x'_u (1-a_{v,u})}{\sqrt{d_v d_u}}$. Add $(u,v)$ to additionCandidates.
```

This is accomplished by the following *Cut Rounding* procedure (Algorithm 4). In both cases, the nodes that are disconnected from the larger component or nodes on the small side of the Fiedler cut are broken off.

In our experiments, the largest fraction of nodes ever disconnected was .5% (including nodes removed due to unbalanced Fiedler cuts).

When the Critical Cut Rounding procedure made the sum of fractional edges crossing the Fiedler cut less than one, the graph would be likely to become disconnected in the later independent rounding step. To prevent this, the final step rounds the largest fractional entry up to 1.

We use randomized rounding (Definition 2.3.1) to round any remaining non-binary entries $a_{u,v}$.

The following theorem by Chung and Radcliffe bounds the resulting spectral perturbations.

Theorem 10. *[Theorem 2 of Chung and Radcliffe (2011)] Let $\mathcal{G}$ be a random graph, generated by including each edge (v,u) independently with probability $a_{v,u}$. Let B be the adjacency matrix of the random graph, and $\delta = \min_i \sum_u a_{v,u}$ the minimum expected degree of any node. For every $\epsilon > 0$, there exists a constant $k = k(\epsilon)$ such that if $\delta > k \ln n$, then with probability at least $1-\epsilon$, all eigenvalues of $L(A)$ and $L(A)$ satisfy $|\lambda_v(L(B)) - \lambda_v(L(A))| \le 3\sqrt{\frac{3\ln(4n/\epsilon)}{\delta}}$.*

We also experimented with using a dependent rounding scheme Gandhi et al. (2006) that includes each edge with probability $a_{v,u}$, but correlates the random choices so that the the number of edges crossing the Fiedler cut exactly matches the expectation (up to fractional parts). Experimentally, we found the performance of both approaches comparable, so we did not include dependent rounding

Algorithm 4: CriticalRound

```
Result: Matrix $A$ with entries $a_{v,u} \in [0,1]$ and $\lambda_k(L(A)) \approx \lambda_k^*$
Input: Matrix $A$ with entries $a_{v,u} \in [0,1]$ ;
Desired spectrum $\lambda^*$ ;
Budget of mass that can be removed from edges crossing cuts $z$;
Closeness threshold $c$ ;
Set cutCorrected flag to False ;
repeat
    Consider the graph of edges $(u, v)$ with $a_{u,v} > 0$. If this graph is disconnected, redefine $A$
      on the entries of the largest connected component ;
    Compute Fiedler cut $\mathcal{S}^*$ with accompanying eigenvectors $\mathbf{x}$ ;
    while $|\mathcal{S}^*| < 5$ do
        Remove $\mathcal{S}^*$ and recompute Fiedler cut $\mathcal{S}^*$ with accompanying eigenvectors $\mathbf{x}$ ;
    end
    if $|\lambda_2^* - \lambda(L(A))| > c$ and the sum of entries crossing $\mathcal{S}^*$ is at least 1 then
        Perform criticalCutRounding (Algorithm 2) ;
        if $A$ is unchanged by criticalCutRounding then
            cutCorrected is True ;
    else
        cutCorrected is True ;
    end
until cutCorrected is True ;
;
If the sum of entries crossing the Fiedler cut is less than 1, round the largest entry crossing
  up to 1 ;
```

in our final algorithm.

7.6 Evaluating Algorithm Components

Our approach utilizes multiple heuristics. To study the contribution of each heuristic to the final result, in Figure 7.1, we plot (for two graphs) the spectra obtained by leaving out various steps. We also compare the spectra against those of simple thresholding approaches as they seem to be utilized by White and Wilson (2007). In addition to the spectra of the input graphs, we show the spectra of six matrices.

1. The result of using the initial template matrix C from the configuration model (with entries $c_{v,u} = d_v d_u / m^*$), and applying our rounding procedure from Section 7.5, without first applying LP (7.1). C is a reasonable candidate for rounding because it matches the degrees of $\mathcal{G}^*$, and all its entries are already in $[0, 1]$.

2. The template matrix output by the LP (7.1). This matrix may have entries outside $[0, 1]$.

3. The matrix output by the LP (7.4), which forced entries from the template inside $[0, 1]$.

4. The final rounded output using the Stiefel manifold optimization along with the LP (7.1) for relaxed spectrum fitting.

5. The final rounded output without using the Stiefel manifold optimization, but only using the LP (7.1) for relaxed spectrum fitting.

6. We derive the following random graph using methods similar to White and Wilson (2007). We combine the spectrum of the target graph and a uniformly random orthonormal matrix X to define the Laplacian matrix $L = X\Lambda^* X'$. A graph is produced by simple thresholding: the edge (v, u) is included iff $l_{v,u} < \theta$ (recall that a Laplacian matrix has negative off-diagonal entries for (v, u) that correspond to edges), with the threshold θ chosen so that the number of edges matches the target.

7. Again in the spirit of White and Wilson (2007), we consider a thresholding of our template matrix $\tilde{A}$. In this case, we include all edges (v, u) with $\tilde{a}_{v,u} > \theta$, again choosing θ such that the number of edges matches the target.

Our experimental results show that the template matrix spectrum closely matches the desired spectrum; thus, the deviation in the final spectrum is mostly a result of pushing the template matrix entries into $[0,1]$, then rounding them. Using C as a template and using our rounding scheme normally suffices to produce strong performance on the spectral *gap*, but fails to match the other eigenvalues. This demonstrates that a careful choice in template helps preserve more of the spectrum. Thresholding the Laplacian and adjacency matrices performs poorly not only on the overall spectrum, but even on the spectral gap. This confirms that a more careful rounding procedure and generation of a suitable template are necessary to generate graphs matching a desired spectrum.

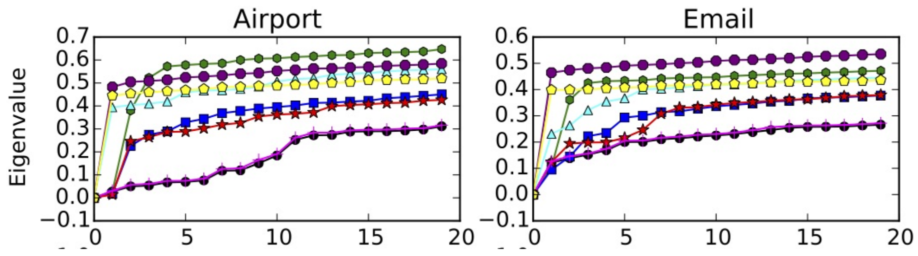


Figure 7.1: Spectra of matrices produced by employing different sets of heuristics. Input graph (black circles), rounding an unfitted matrix (green hexagons), fitted template matrix (magenta plus), fractional graph (cyan triangles), output graph without Stiefel (blue squares), output graph with Stiefel (red stars), thresholding the Laplacian (yellow pentagons), thresholding the adjacency matrix (purple octagons).

We observe that the Stiefel manifold optimization did not always improve performance of the final result, but more often than not provided small improvements. However, for large graphs, the computation becomes expensive.

Chapter 8

Generating graphs with deep random walks

The behavior of random walks is determined by the connectivity structure of the graph (Section 5.0.1)[1]. For example, how many steps it takes for a walk to leave a cut tells us something about how connected the cut is to the rest of the graph (Spielman and Teng, 2013). This chapter explores how to build random graphs from *finite* random walks drawn from target graph $\mathcal{G}^*$ of a fixed length using a neural network. NetGAN by Bojchevski et al. (2018) trains a generative adversarial network (GAN)[2] to generate sequences of nodes that resemble walks on $\mathcal{G}^*$. These synthetic walks are then used to construct a random $\mathcal{G}$ by (1) counting the number of times edges appear on the walks and (2) including edges in $\mathcal{G}$ that appear enough times in the synthetic walks. By using synthetic walks instead of real walks to construct $\mathcal{G}$, variation is introduced into the distribution over $\mathcal{G}$ induced by the edge counts. If edge counts were used in the same way to contruct $\mathcal{G}$ using the real walks, only edges in $\mathcal{G}^*$ could appear in $\mathcal{G}$ and $\mathcal{G}$ would also be a subgraph of $\mathcal{G}^*$. Therefore, the GAN acts as a *noise generator* to introduce variation into $\mathcal{G}$.

We investigate how NetGAN can be used to learn the connectivity structure of $\mathcal{G}^*$. While the long-term behavior of random walks encodes global structure, it is not clear how much global structure can be learned by training with relatively short walks (length 16 is used by Bojchevski et al. (2018)). Are there variations of NetGAN that make learning global structure possible? These short walks act as "semi-local" features: not fully local (like edges) nor global (like long term random walk behavior or the entire graph). We are interested if global properties can be learned under this paradigm where we learn a distribution over semi-local features and then combine these features

[1]Random walks are defined in Section 2.4

[2]Generative adversarial networks are defined in Section 4.5.1.

to construct graphs that are globally similar through both the existing NetGAN and new variants that target this goal. More generally, Bojchevski et al. (2018) shows that the NetGAN can compete with a number of benchmarks on a variety of metrics. We are interested in what algorithmic choices contribute most to that success. We conduct our investigation of the NetGAN to this end.

8.1 NetGAN algorithm: Learning a random walk distribution and sampling graphs

In this section we describe NetGAN by Bojchevski et al. (2018).

8.1.1 Deep learning with a single example

One fundamental difference between traditional random graph models and the more recent deep generative graph networks is that they are designed to train from a large data set (Section 4.5). The deep methods are often trained to (1) learn a model that explains shared features among graphs in the data set and (2) produce a distribution over graphs so that the graphs with high probability have these learned features. The loss functions are designed to penalize models that memorize features specific to only a few graphs and do not generalize to the entire set. However, if there is only one graph in the data set then alternative loss functions would be necessary because there are no counterexamples to any features memorized.

One approach to learning from a single graph $\mathcal{G}^*$ is to learn a distribution over graph features instead of over graphs. The training data can then be features of $\mathcal{G}^*$. If the feature set is large, this avoids the problem of learning from a single data point. The goal would be to learn from samples drawn from a distribution p^* over some features of $\mathcal{G}^*$. We can sample empirically from p^* and then use deep learning to learn a distribution $p \approx p^*$ from examples as we normally would. This approach relies on:

1. Choosing informative graph features to define p^* over. This thesis emphasizes the importance of features that describe the connectivity of a graph. For what connectivity features can we build a "useful" distribution and how would we sample from it?

2. A method to learn p from training data drawn from p^*. How will the graph features be

represented in a form that a deep neural network can understand? Usually, neural networks learn from *Euclidean* data. If we represent graph features with Euclidean data, can we avoid a dimension blow-up if the possibilities for each feature is exponential in the number of vertices?

3. A method to construct random graph $\mathcal{G}$ from an ensemble of features drawn from p. If we draw a collection of graph features from a distribution, there might not be a graph with all of these graph features. Should each feature be drawn independently? How should these features be combined to make a graph? Will the features prescribe the graph exactly, or will there be additional choices?

8.1.2 NetGAN: generating graphs from random walks drawn from a single graph

Bojchevski et al. (2018) designs the *NetGAN* and is the first to design a graph generator this way (that we know of) that trains a deep model to first generate random features (random walks) and then generate graphs resembling a *single* input. At a high level, NetGAN trains a GAN to generate walks that are similar to walks sampled from $\mathcal{G}^*$. In order to generate graphs from these walks, a matrix $\tilde{A}$ is constructed with the v, u-th entry equal to the number of times vertices v and u appear consecutively on the same walks. Lastly, m^* edges are added to the resulting graph $\mathcal{G}$ using an iterative procedure that (1) makes sure each node has at least one edge and (2) includes edges proportional to $\tilde{a}_{v,u}$. Algorithm 5 describes NetGAN in additional detail.

One motivation for training the GAN with random walks over other graph features is that they strike a middle ground between being fully local (like individual edges) vs. global (like the entire graph)[3]. The other advantage of using random walks as features is that it avoids creating a graph representation the GAN can understand and reduces the problem to creating a node representation. A node representation for nodes in a graph of size n appears easier to accommodate than a graph representation because there are only a linear number of nodes but a quadratic number of edges that can appear in the graph. The walks are then vectors of nodes representations.

The idea behind NetGAN is that learning to reproduce random walks drawn from $\mathcal{G}^*$ well enough, but not perfectly, should result in implicitly learning a fairly accurate representation of the

[3]Discussion about difference between local and global features in Section 1.3.

key graph features. The true distribution p^* is a random walk distribution. It is critical that the GAN not learn to reproduce the random walks perfectly because if it does, then all synthetic walks drawn from the generator will be along edges $(v, u) \in \mathcal{E}^*$. In the graph construction phase, NetGAN includes edge (v, u) in $\mathcal{G}$ with non-zero probability if and only if it appears with non-zero frequency along the walks. Because the only edges that appear with non-zero frequency on walks drawn from p^* are edges in $\mathcal{E}^*$, all output graphs $\mathcal{G}$ will be sub-graphs of $\mathcal{G}^*$. Instead of relying on exclusively on edge deletions to provide diversity into the samples, NetGAN uses an early stopping criterion to stop training before the GAN finishes learning p^*. By stopping training early, the learned distribution traversed pairs $(v, u) \notin \mathcal{E}^*$.

In the first phase of the NetGAN algorithm, integer sequences $(v_1, v_2, \ldots, v_k)$ where $v_t \in \mathcal{V}^*$ are generated from a partially trained GAN that represent random walks on $\mathcal{G} = (\mathcal{V}^*, \mathcal{E})$. We note here that $\mathcal{V}^* = [n^*]$ so that each vertex can can be represented as an integer, which is important to how the GAN will represent vertices which we explain further in Section 8.1.3. The second phase of the NetGAN algorithm is using synthetic walks from the GAN generator to construct $\mathcal{G}$.

The GAN is trained to learn "random walk" sequences that are similar to random walks on $\mathcal{G}^*$[4]. To detect overfitting, the random walk data the GAN is trained on is along a subset of $\mathcal{E}^*$ and a disjoint hold-out set is used to detect if the GAN is memorizing. Edge set $\mathcal{E}^*$ is split into three parts: training set $\mathcal{E}_r$, validation set $\mathcal{E}_v$ and testing set $\mathcal{E}_t$ of size $(1 - q_v - q_s)m^*$, $q_v m^*$ and $q_s m^*$ respectively where $0 < q_s + q_v < 1$. To generate training walk data for the GAN, walks are performed along a new graph $\mathcal{G}_r = (\mathcal{V}^*, \mathcal{E}_r)$.

8.1.3 GAN architecture

The GAN is built using two *Recurrent Neural Networks* (RNN): one for the *generator* and one for the *discriminator* (Section 4.5.1). A RNN can be thought of as k copies of the same neural network. Each copy is run in sequence and the t-th copy uses part of the $(t-1)$-th output as part of its input. In this way, the RNN "memorizes" part of what it has already seen and uses its memory to compute the next output. The GAN uses a special type of RNN called a *Long-Short Term Memory* machine Hochreiter and Schmidhuber (1997). More formally, a RNN maps two vectors, a memory state $\boldsymbol{m}^{(t)}$

[4]Random walks, as defined in Section 2.4, are defined on graphs with consecutive vertices walking along edges. The GAN generates integers that represent vertices, however, subsequent integers need not represent an edge in any graph.

Algorithm 5: NetGAN algorithm.

```
Result: $\mathcal{G}$
Input: $\mathcal{G}^*$;
*Random walk parameters*: random walk algorithm $\mathcal{A}$;
Random walk length $k$;
Number of evaluation transitions $T$;
Batch size $b$;
*Hold out set parameters:* validation set size $0 < q_v < 1$ ;
test set size $0 < q_t < 1 - q_v$ ;
*Stopping parameters*: stoppingCondition ;
*Number of training steps before evaluation*: $\ell$;
Sample subsets $\mathcal{E}_v$ and $\mathcal{E}_t$ of size $q_v m$ and $q_t m$ from $\mathcal{E}^*$ without replacement.
 $\mathcal{E}_r = \mathcal{E}^* \setminus (\mathcal{E}_v \cup \mathcal{E}_t)$ ;
$\mathcal{G}_r = (\mathcal{V}^*, \mathcal{E}_r)$ ;
Construct the GAN using $\mathcal{A}$, $\mathcal{G}_r$, $b$, $k$;
$\tilde{A} = 0_{n^*}$ ;
while *stoppingCondition not met* do
    Take $\ell$ GAN training steps (Algorithm 8) ;
    Sample $T/(k-1)$ walks $\mathcal{W}$ from the GAN generator ;
    $\tilde{B} = 0_{n^* \times n^*}$ ;
    for $i \leftarrow 1$ to *T/(k- 1)* do
        $w = \mathcal{W}_i$ ;
        for $j \leftarrow 1$ to *k- 1* do
            $u = \boldsymbol{w}[j]$, $v = \boldsymbol{w}[j+1]$ ;
            $\tilde{b}_{v,u} = \tilde{b}_{v,u} + 1$ ;
            $\tilde{b}_{v,u} = \tilde{b}_{v,u} + 1$ ;
        end
    end
    Evaluate stoppingCondition on $\tilde{B}$;
    If $\tilde{B}$ has higher utility than $\tilde{A}$, $\tilde{A} = \tilde{B}$;
end
Generate $\mathcal{G}$ with $m^*$ edges using *fixed-edge* graph sampling which takes $\tilde{A}$ as input
 (Algorithm 12);
```

	RNN	Random Walk
Vertex	One-hot vector $\boldsymbol{x}_v^{n^*}$	Vertex $v \in \mathcal{V}^* = [n^*]$
State	Vector $\boldsymbol{v}^{(t)} \in \{\boldsymbol{x}_i^{n^*}\}$	The t-th vertex $\boldsymbol{w}[t]$ in a random walk $\boldsymbol{w}$ on a graph with $\mathcal{V}^*$
Memory	Vector $\boldsymbol{m}^{(t)} \in \mathbb{R}^{s_g}$	For random walk of order q, the q previous vertices $\boldsymbol{w}[t-1], \ldots, \boldsymbol{w}[t-q]$ in $\boldsymbol{w}$
Distribution over states	Vector $\boldsymbol{p}^{(t)} \in \mathbb{R}^{n^*}$ which is normalized to form a distribution over $[n^*]$	Distribution from which the value of $\boldsymbol{w}[t]$ is drawn

Table 8.1: Correspondence between RNN and random walk components.

and an input $\boldsymbol{u}^{(t)}$, to two vectors, the next memory state $\boldsymbol{m}^{(t+1)}$ and an output $\boldsymbol{p}^{(t+1)}$. To see how the generator RNN is used to generate sequences that can be interpreted as random walks, $\boldsymbol{p}^{(t)}$ is a n^*-dimensional vector and used to construct a probability distribution over $[n^*]$. The state $\boldsymbol{v}^{(t)}$ is drawn randomly according to $\boldsymbol{p}^{(t)}$ and can be thought of as the t-th vertex in the random walk. The correspondence between the RNN and random walk components is listed in Table 8.1.

State $\boldsymbol{v}^{(t)}$ is a *one-hot* n^*-dimensional vector where $\boldsymbol{x}_i^n$ is the i-th n-dimensional one-hot vector with $(x_i^n)_i = 1$ and $(x_i^n)_j = 0$ for all $i \neq j$. Vertices $\mathcal{V}^* = [n^*]$ so there is a correspondence between node $v \in \mathcal{V}^*$ and $\boldsymbol{x}_v^{n^*}$. To help accommodate large n^*, $\boldsymbol{v}^{(t)}$ is projected to a n'-dimensional input vector $\boldsymbol{u}^{(t)}$ with $n' < n^*$ using a projection function $g'_{\boldsymbol{\gamma}_g}(\boldsymbol{v}^{(t)})$ where $\boldsymbol{\gamma}_g$ is learned. The RNN generator is written $g_{\boldsymbol{\theta}_g}(\boldsymbol{m}^{(t)}, \boldsymbol{u}^{(t)})$ where $\boldsymbol{\theta}_g$ is a learned vector. State $\boldsymbol{v}^{(t)}$ takes value $\boldsymbol{x}_v^n$ for $v \in [n^*]$ with probability $p_v^{(t)}/Z$ where $Z = \sum_{v=1}^{n^*} p_v^{(t)}$. The initial $\boldsymbol{m}^{(0)}$ is sampled from mapping a random Gaussian noise vector to an s_g-dimensional vector. The parameters $\boldsymbol{\nu}_g$ of noise mapping $g''_{\boldsymbol{\nu}_g}$ are also learned. Now the first copy of the RNN is called with $g_{\boldsymbol{\theta}_g}(\boldsymbol{m}^{(0)}, \mathbf{0}_{m^*}) = (\boldsymbol{m}^{(1)}, \boldsymbol{p}^{(1)})$ where $\boldsymbol{v}^{(1)}$ is sampled according to $\boldsymbol{p}^{(1)}$. After k calls, we have a sequence $\boldsymbol{v}^{(1)}$,$\boldsymbol{v}^{(2)}$,$\ldots$,$\boldsymbol{v}^{(k)}$ which can be interpreted as a walk $v_1, v_2, \ldots, v_k$.

The discriminator is also built using an RNN $f_{\boldsymbol{\theta}_d}$ parameterized by learned parameters θ_d. At each time step t, the discriminator (1) reads in the one-hot vector $\boldsymbol{v}^{(t)}$ and projects it down to $f'_{\boldsymbol{\gamma}_d}(\boldsymbol{v}^{(t)}) = \boldsymbol{u}^{(t)}$ (2) passes $\boldsymbol{u}^{(t)}$ to the RNN and outputs $f_{\boldsymbol{\theta}_d}(\boldsymbol{m}^{(t)}, \boldsymbol{u}^{(t)}) = (\boldsymbol{m}^{(t+1)}, \boldsymbol{o}^{(t+1)})$. The last output state $\boldsymbol{o}^{(k)}$ is mapped to a scalar $o^{(k)} = f''_{\boldsymbol{\eta}_d}(\boldsymbol{o}^{(k)})$ which we can think of as a score over the sequence $\boldsymbol{v}^{(1)}, \boldsymbol{v}^{(2)}, \ldots, \boldsymbol{v}^{(k)}$ which is the discriminator scoring the walk $v_1, v_2, \ldots, v_k$.

8.1.4 Training the Random Walk GAN

We now introduce the loss functions for training the generator parameters $\hat{\boldsymbol{\theta}}_g = (\boldsymbol{\theta}_g, \boldsymbol{\gamma}_g, \boldsymbol{\nu}_g)$ and the discriminator parameters $\hat{\boldsymbol{\theta}}_d = (\boldsymbol{\theta}_d, \boldsymbol{\gamma}_d, \boldsymbol{\eta}_d)$. We write the final discriminator output $o^{(k)}$ as a function of a walk $w = v_1, v_2, \dots, v_k$: $F_{\hat{\boldsymbol{\theta}}_d}(w) = o^{(k)}$. Parameters $\hat{\boldsymbol{\theta}}_g$ and $\hat{\boldsymbol{\theta}}_d$ are trained using the Wasserstein loss Arjovsky et al. (2017); Gulrajani et al. (2017)[5]. In each iteration, a batch of b "real" walks $\mathcal{W} = \boldsymbol{w}_1, \dots, \boldsymbol{w}_b$ are generated from the actual random walk process, and another batch of b "fake" walks $\mathcal{W}(\hat{\boldsymbol{\theta}}_g) = \boldsymbol{w}'_1, \dots, \boldsymbol{w}'_b$ are generated by the generator (Algorithms 7,6). The average output on real/fake walks is $a(\mathcal{W}, \hat{\boldsymbol{\theta}}_d) = \frac{1}{b}\sum_{i=1}^{w} F_{\hat{\boldsymbol{\theta}}_d}(\boldsymbol{w}_i)$, $a(\mathcal{W}(\hat{\boldsymbol{\theta}}_g), \hat{\boldsymbol{\theta}}_d) = \frac{1}{b}\sum_{i=1}^{w} F_{\hat{\boldsymbol{\theta}}_d}(\boldsymbol{w}'_i)$, and the Wasserstein objective for the discriminator is to minimize $\mathcal{L}(\hat{\boldsymbol{\theta}}_d) = a(\mathcal{W}(\hat{\boldsymbol{\theta}}_g), \hat{\boldsymbol{\theta}}_d) - a(\mathcal{W}, \hat{\boldsymbol{\theta}}_d)$ over choices of $\hat{\boldsymbol{\theta}}_d$. Thus, the goal of the discriminator is to label $\boldsymbol{w}'_i \in \mathcal{W}(\hat{\boldsymbol{\theta}}_g)$ as fake and $\boldsymbol{w}_i \in \mathcal{W}$ as real. The Wasserstein objective for the generator is to minimize $\mathcal{L}(\hat{\boldsymbol{\theta}}_g) = -a(\mathcal{W}(\hat{\boldsymbol{\theta}}_g), \hat{\boldsymbol{\theta}}_d)$ over choices of $\hat{\boldsymbol{\theta}}_g$ (which are implicitly used in the generative process for the $\boldsymbol{w}'_i$), i.e., to minimize the number of fake walks labeled as fake by the discriminator. Parameters $\hat{\boldsymbol{\theta}}_g$ and $\hat{\boldsymbol{\theta}}_d$ are updated using stochastic batch gradient descent. Details are in Algorithm 8.

```
Algorithm 6: GenerateRealWalks
  Result: Walks drawn from $\mathcal{G}_r$ using $\mathcal{A}$: $\boldsymbol{w}_1, \boldsymbol{w}_2, \dots \boldsymbol{w}_b$ ;
  Input: Number of walks $b$;
  Walk length $k$;
  Walk algorithm $\mathcal{A}$;
  Training graph $\mathcal{G}_r$;
  Draw $b$ real random walks $\boldsymbol{w}_1, \boldsymbol{w}_2, \dots \boldsymbol{w}_k$ of length $k$ using random walk algorithm $\mathcal{A}$ that
    takes $\mathcal{G}_r$, $b$, $k$ as arguments. ;
```

```
Algorithm 7: GenerateFakeWalks
  Result: Synthetic walks drawn from a generator parameterized by $\hat{\boldsymbol{\theta}}_g$: $\boldsymbol{w}'_1, \boldsymbol{w}'_2, \dots \boldsymbol{w}'_b$
  Input: Generator parameters $\hat{\boldsymbol{\theta}}_g$;
  Number of walks $b$;
  Length of walks $k$;
  Draw $b$ fake random walks $\boldsymbol{w}'_1, \boldsymbol{w}'_2, \dots \boldsymbol{w}'_k$ of length $k$ using generator parameterized by $\hat{\boldsymbol{\theta}}_g$;
```

NetGAN can be trained using any *random walk algorithm* $\mathcal{A}$ (Section 2.4). A random walk algorithm $\mathcal{A}$ takes $\mathcal{G}$, some set of parameters θ, and a number of random walks k. It produces k random walks on $\mathcal{G}$ according to the walk specified by θ. The *standard random walk* is Markovian

[5]Wasserstein loss defined in Section 4.5.1

Algorithm 8: One GAN training round

```
Result: Updated generator and discriminator parameters $\hat{\boldsymbol{\theta}}_g$ and $\hat{\boldsymbol{\theta}}_d$
Input: Generator parameters $\hat{\boldsymbol{\theta}}_g$;
Discriminator parameters $\hat{\boldsymbol{\theta}}_d$;
Number of discriminator iterations $\ell_d$ ;
Number of generator iterations $\ell_g$ ;
Number of walks $b$;
Length of walk $k$;
Random walk algorithm $\mathcal{A}$;
Training graph $\mathcal{G}_r$;
for $i \leftarrow 1$ to $\ell_g$ do
    $\mathcal{W}(\hat{\boldsymbol{\theta}}_g)$ = GenerateFakeWalks($\hat{\boldsymbol{\theta}}_g$,$b$,$k$) (Algorithm 7) ;
    $\mathcal{L}(\hat{\boldsymbol{\theta}}_g) = -\frac{1}{b}\sum_{w'_i \in \mathcal{W}(\hat{\boldsymbol{\theta}}_g)} F_{\hat{\boldsymbol{\theta}}_d}(w'_i)$ ;
    Update $\hat{\boldsymbol{\theta}}_g$ using stochastic gradient descent ;
end
for $i \leftarrow 1$ to $\ell_d$ do
    $\mathcal{W}$ = GenerateRealWalks($\mathcal{A}$,$\mathcal{G}_r$,$b$,$k$) (Algorithm 6) ;
    $\mathcal{W}(\hat{\boldsymbol{\theta}}_g)$ = GenerateFakeWalks($\hat{\boldsymbol{\theta}}_g$,$b$,$k$) (Algorithm 7) ;
    $\mathcal{L}(\hat{\boldsymbol{\theta}}_d) = \frac{1}{b}\left(\sum_{w'_i \in \mathcal{W}(\hat{\boldsymbol{\theta}}_g)} F_{\hat{\boldsymbol{\theta}}_d}(w'_i) - \sum_{w_i \in \mathcal{W}} F_{\hat{\boldsymbol{\theta}}_d}(w_i)\right)$;
    Update $\hat{\boldsymbol{\theta}}_d$ using stochastic gradient descent ;
end
```

and we write its transition matrix as Q (for more on Markovian random walks, see Section 2.4). The random walk algorithm for sampling Markovian random walks is in Algorithm 9, we introduce another Markovian random walk in Section 8.3.

NetGAN Bojchevski et al. (2018) uses the *node2vec* random walk algorithm. Node2vec samples from a second-order Markov chain parameterized by two parameters z_1, z_2 in $[0, 1]$ that at step t, control preference toward or against returning to $\boldsymbol{w}[t-1]$ (Grover and Leskovec, 2016). If z_1 is small compared to z_2, the walk will be biased toward returning to $\boldsymbol{w}[t-1]$, inducing a BFS like exploration of the graph. Alternatively, if z_1 is large compared to z_2, the walk will be biased against returning to $\boldsymbol{w}[t-1]$, inducing a DFS like exploration of the graph. If $z_1 = z_2 = 1$, then the walk is exactly the standard random walk which treats all neighbors (including $\boldsymbol{w}[t-1]$) equally. Experiments show that setting the parameters to anything except $z_1 = z_2 = 1$ (which results in exactly the standard random walk) has little impact on the performance; we therefore will focus only on the standard random walk in our experiments. The node2vec random walk algorithm is shown in Algorithm 10. One reason the node2vec parameters may make little impact on performance is any

Algorithm 9: Sampling Markovian random walks

```
Result: Walks on $\mathcal{G}_r$ sampled using Markov chain defined by $Q$: $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots \boldsymbol{w}_b$ ;
Input: Training graph $\mathcal{G}_r$;
Number of walks $b$;
Length of walks $k$;
Transition matrix $Q$;
$\pi$ is the stationary vector of $Q$ so that $\pi Q = \pi$ ;
for $i \leftarrow 1$ to $b$ do
    $\boldsymbol{w}_i[1] = u$ with probability $\pi_u$ ;
    for $j \leftarrow 2$ to $k$ do
        $\boldsymbol{w}_i[j] = u$ with probability $q_{u,\boldsymbol{w}_i[j-1]}$ ;
    end
end
```

Algorithm 10: node2vec random walk

```
Result: Walks sampled from $\mathcal{G}_r$: $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots \boldsymbol{w}_b$
Input: Training graph $\mathcal{G}_r$;
Number of walks $b$;
Length of random walk $k$;
Node2vec parameters $z_1$, $z_2$
for $i \leftarrow 1$ to $b$ do
    $\boldsymbol{w}_i[1] = v$ with probability $1/|\mathcal{V}^*|$ ;
    $\boldsymbol{w}_i[2] = u$ with probability $1/d_{\boldsymbol{w}_i[1]}$ for all neighbors $u$ of $\boldsymbol{w}_i[1]$ ;
    for $j \leftarrow 3$ to $k$ do
        for all neighbors $v$ of $\boldsymbol{w}_i[j]$ do
            If $v = \boldsymbol{w}_i[j-1]$, $\alpha(v) = 1/z_1$ ;
            If $v$ is a neighbor of $\boldsymbol{w}_i[j-1]$, $\alpha(v) = 1$ ;
            Else, $\alpha(v) = 1/z_2$ ;
            Let $Z$ be the sum of $\alpha(v)$ over all neighbors $v$ of $\boldsymbol{w}_i[j]$ ;
            $\boldsymbol{w}_i[j] = v$ with probability $\alpha(v)/Z$ ;
        end
    end
end
```

ordering induced by bias parameters is omitted in the encoding of $\mathcal{W}$ into $\tilde{A}$. Our graph generator is built from $\tilde{A}$, therefore, we turn toward using random walks that induce probability distributions over visiting edges that will make $\tilde{A}$ more descriptive of the graphs we want to sample (Section 8.3).

NetGAN avoids overfitting by using edge prediction on a held-out validation set $\mathcal{E}_v \subset E^*$. Another negative set $\tilde{\mathcal{E}}_v$ of non-edges (i.e., $\tilde{\mathcal{E}}_v \cap E^* = \emptyset$) is used to test false negatives. We compute the ability of $\tilde{A}$ to predict edges by assigning scores $\tilde{a}_{v,u}$ to each $(v, u) \in \mathcal{E}_v \cup \tilde{E}_v$. Edge prediction is then measured using ROC AUC and AP (explained below) Powers (2011). The edge prediction stopping criterion is evaluated once every ℓ GAN training steps and records the sum of the ROC AUC and AP scores. NetGAN stops training if the sum of the ROC AUC score and AP scores does not get larger after z sequential evaluations.

Algorithm 11: Edge Prediction Stopping Criterion

```
Result: Yes if edge prediction has not improved for $z$ number of iterations and no
        otherwise. Matrix $\tilde{A}$ is the frequency matrix with the best edge prediction seen.
Input: Positive validation edge set $\mathcal{E}_v$;
Negative validation edge set $\tilde{\mathcal{E}}_v$;
Most recent model $\tilde{B}$;
Highest edge prediction seen best ;
Number of evaluations left before terminating evalsLeft ;
Tolerance for no improvement $z$;
Frequency matrix with best edge prediction seen $\tilde{A}$
True scores $\mathbf{x} = \{1 \text{ for } (v,u) \in \mathcal{E}_v, 0 \text{ for } (v,u) \in \tilde{\mathcal{E}}_v\}$ ;
Valid scores $\mathbf{y} = \{\tilde{b}_{v,u} \text{ for } (v,u) \in \mathcal{E}_v \cup \tilde{\mathcal{E}}_v\}$ ;
If $\text{AUC}(\mathbf{x},\mathbf{y}) + \text{AP}(\mathbf{x},\mathbf{y}) \leq$ best then evalsLeft = evalsLeft $- 1$;
Else evalsLeft = $z$, *best* = $\text{AUC}(\mathbf{x},\mathbf{y}) + \text{AP}(\mathbf{x},\mathbf{y})$, $\tilde{A} = \tilde{B}$ ;
If evalsLeft == 0, return Yes ;
Else, return No ;
```

AUC ROC and AP Powers (2011)

The *Receiving Operatic Characteristic* ROC curve depicts the ability of threshold binary classifiers to correctly label one-dimensional data $\mathbf{y}$ using true binary labels $\mathbf{x}$. ROC curve is $(\text{FPR}(t), \text{TPR}(t))$

where FPR(t)/TPR(t) denote the False/True positive rates of classifying all $y_i \leq t$ as 0 and $y_i > t$ as 1 for all thresholds t. The ROC AUC describes the area under the curve. Linearly separable $\mathbf{y}$ separable by t^* will have ROC AUC equal to 1. To see this, for all thresholds $t \leq t^*$, $\text{FPR}(t) = 0$. For all $t \geq t^*$, $\text{TPR}(t) = 1$. Thus, the area under the curve describes the area of the unit square.

The *Average Precision* is the area under the Precision-Recall curve which plots the precision of binary classifiers on one dimensional data $\mathbf{y}$ using true binary labels $\mathbf{x}$. The Precision-Recall curve are points $(\text{R}(t), \text{Pr}(t))$ where recall R(t) is the fraction of values $x_i = 1$ that are labeled 1 by threshold t (indices i with $y_i > t$) and the precision Pr(t) is the fraction of values $y_i > t$ that have true labels 1 (indices i with $x_i = 1$). Again, if $\mathbf{y}$ is linearly separable by t^* then for values $t < t^*$, recall will be 1. For values $t > t^*$, precision will be 1 so the average precision is 1.

8.1.5 Generating a graph using random walk samples

Algorithm 12: Fixed-Edge graph sampling

```
Result: Graph $\mathcal{G}$
Input: Frequency matrix $\tilde{A}$;
Number of target edges $m^*$;
Target nodes $\mathcal{V}^*$;
Initialize $\mathcal{E} = \emptyset$ ;
$m' = 0$ ;
Permute $\mathcal{V}^*$ with uniformly random permutation $\pi$ ;
for $i \leftarrow 1$ to $n^*$ do
    $v = \pi(i)$ ;
    Sample random vertex $u$ so that $u$ is drawn with probability $\tilde{a}_{v,u}/Z$ where
      $Z = \sum_{u=1}^{n^*} \tilde{a}_{v,u}$ ;
    If $(v,u)$ not in $\mathcal{E}$, add $(v,u)$ to $\mathcal{E}$ and $m' = m' + 1$. Else, move to next vertex ;
end
Sample $m^* - m'$ pairs $(v,u)$ without replacement from $\overline{\mathcal{E}} = \mathcal{V}^* \times \mathcal{V}^* \setminus \mathcal{E}$ from so $(v,u)$ is
  included with probability $\frac{\tilde{a}_{v,u}}{Z}$ where $Z = \sum_{(v,u)\in\overline{\mathcal{E}}} \tilde{a}_{v,u}$ ;
Add all sampled pairs to $\mathcal{E}$;
Return $\mathcal{G} = (\mathcal{V}^*, \mathcal{E})$.
```

Once the GAN has been sufficiently trained, NetGAN samples a set $\mathcal{W}(\hat{\boldsymbol{\theta}}_g)$ of size c of k-step walks. From $\mathcal{W}(\hat{\boldsymbol{\theta}}_g)$, it constructs a frequency matrix $\tilde{A}$, with $\tilde{a}_{v,u}$ being the number of times the edge (v, u) (in either direction) appears in walks in $\mathcal{W}(\hat{\boldsymbol{\theta}}_g)$; if (v, u) appears multiple times in the same walk, it is counted multiple times. NetGAN constructs an output graph $\mathcal{G}$ with exactly m^* edges on $\mathcal{V}^*$ using a method we call *fixed-edge iterative sampling (FE)*. Fixed-edge iterative sampling goes through $\mathcal{V}^*$ in uniformly random order; when it is node v's turn, one of its incident edges (v, u) is sampled with probability proportional to $\tilde{a}_{v,u}$. After this step, some $m' \leq n^*$ edges have been added to $\mathcal{E}$ and each node is incident on at least one edge. The remaining $m^* - m'$ edges are drawn with probabilities proportional to $\tilde{A}vu$ without replacement. The main reason for sampling at least one edge for each node is to make disconnected graphs less likely. Since the FMMC also serves this purpose, we consider a much simpler, independent, graph sampling in Section 8.5.

8.2 Our contribution

We conduct a principled investigation of the NetGAN and it's different components guided by the fact that the utility of the random walk generator is entirely in the frequency matrix $\tilde{A}$ that it provides for graph generation (Observation 1). Critical is the fact that if random walk GAN learns $p = p^*$, then the only edges with probability mass are included in $\mathcal{E}^*$ and the edges sampled will strictly be subsets of $\mathcal{E}^*$. Instead, we aim to learn $p \approx p^*$. How the GAN is trained and how edge probabilities are derived from sampled random walks (and thus p) will be crucial in deciding how close does p need to be to p^* in order to ensure graph structure is learned while not over-fitting to the target graph. We point out that while over-fitting is a concern for most distribution learning tasks, in this case we are concerned with over-fitting to the *true* distribution (the distribution from which the random walks are generated) instead of the empirical which makes this a special case.

We focus on the following questions around the NetGAN components:

1. How important or useful is the specific choice of random walk (Section 8.3)? Bojchevski et al. (2018) use the node2vec walk with second-order memory Grover and Leskovec (2016) for generating random walks. Could other random walk distributions provide better matches of large-scale structural properties? In particular, if $\mathcal{G}^*$ has several sparsely connected components, it may be desirable to over sample edges across those cuts; one can accomplish this by using

the fastest mixing Markov Chain (FMMC) for $\mathcal{G}^*$ under the uniform stationary distribution Boyd et al. (2004).

2. How important is the *length* of the training walks? The frequency of individual edges is not affected by the length of the walks, so any differences must be the result of making the task of the generator or discriminator harder or easier (Section 8.4).

3. What termination condition should be used for training the GAN? If the generator can perfectly memorize the distribution, then diversity will be missing among the graphs $\mathcal{G}$. Bojchevski et al. (2018) used a termination criterion based on edge prediction for a hold-out set, and terminated the training before convergence. What is the impact of other criteria on the output graphs $\mathcal{G}$ (Section 8.6)?

4. How does the GAN noisily map A^* to $\tilde{A}$? If the noise were simply uniform over all pairs not included in the random walks, then using a GAN would be unnecessary — one could use a uniform generator in place of the GAN (Section 8.8).

5. How should the graph $\mathcal{G}$ be generated from $\tilde{A}$? How important is it to match m^* exactly, rather than in expectation? We evaluate the impact of including each edge (u, v) independently in $\mathcal{G}$ with probability essentially proportional to $\tilde{a}_{u,v}$ (Section 8.5).

The main results of our experimental evaluation are (1) an improved termination condition also considering the expected spectrum of the generated graph unambiguously makes the output graphs more similar to the input graphs; however, for some graphs, the added training comes at a diversity cost; (2) using the FMMC helps keep the output graphs connected in the presence of sparse cuts, but can hurt various other graph properties; (3) independent inclusion of edges produces graphs just as similar to the input graph as keeping the number of edges fixed; (4) the length of the random walks used for training is important; shorter walks or a memoryless GAN seem to make training slower or impossible; (5) larger GANs can overfit when the models are trained for a long time; (6) the GAN adds less noise (compared to uniform noise) across the (sparse) *Fiedler cut*, causing the Fiedler cut to remain sparse in the generated graphs.

8.3 Fastest Mixing Markov Chain

For the big-picture goal of producing graphs $\mathcal{G}$ whose structure is "similar" to $\mathcal{G}^*$, some edges may be more critical than others, and should therefore be sampled with probability larger than $\frac{1}{2}\ m^*$. As an extreme example, consider a *barbell* graph, consisting of two cliques $K_{n/2}$, connected by a single edge. To prevent the output graphs $\mathcal{G}$ from becoming disconnected too often, the training walks should oversample the connecting edge, whereas the exact identity of edges that are sampled within the cliques is less important.

It is well known, e.g., Levin and Peres (2017); Motwani and Raghavan (1990), that there is a close connection between the mixing time of a Markov chain, the sparsity of cuts in the graph, and the spectral gap of the chain's transition matrix (Theorem 2). In particular, given a fixed underlying graph, Markov chains that mix rapidly, i.e., have large spectral gap, will place higher weight on edges across sparse cuts. This motivates our approach of using the fastest mixing Markov chain (FMMC) subject to the uniform stationary probability distribution as a natural way to oversample important edges.

A natural concrete objective is to assign symmetric non-negative edge weights R so as to maximize the second-largest eigenvalue modulus (SLEM) $\mu(R) = \max_{i=2,\ldots,n} |\lambda_i(R)| = \max\{\lambda_2(R), -\lambda_n(R)\}$ of the random walk matrix. Boyd et al. (2004) show that minimizing SLEM can be formulated as a convex program, and that standard primal-dual interior-point methods exactly minimize SLEM for instances with up to roughly 1000 edges. For larger graphs, also following Boyd et al. (2004), we use *subgradient* methods Overton and Womersley (1993) to approximately compute the FMMC.

Next, we derive some intuition why the fastest-mixing Markov chain gives preference to edges across sparse (bottleneck) cuts. From Cheeger's inequality, we know that the second smallest eigenvalue is related to the connectivity across a sparse cut (Theorem 1). The fastest-mixing Markov chain uses the second smallest eigenvalue (and largest) and its associated *Fiedler cut* to place high transition probability across sparse cuts so that the walk will mix quickly. Recall that by the variational characterization of eigenvalues Horn and Johnson (1990), we can write $\lambda_2(R)$ and $-\lambda_n(R)$ as follows:

$$\begin{aligned}\lambda_2(R) &= \sup\{\mathbf{x}^T R\mathbf{x} \mid ||\mathbf{x}||_2 \le 1, \mathbf{1}^T\mathbf{x} = 0\},\\ -\lambda_n(P) &= \sup\{-\mathbf{x}^T R\mathbf{x} \mid ||\mathbf{x}||_2 \le 1\},\end{aligned}$$

Expanding $\mathbf{x}^T R\mathbf{x} = \sum_{v,u} x_v x_u r_{v,u}$ shows that for a given chain R, the entries of the eigenvector $\mathbf{x}_2$ corresponding to $\lambda_2(R)$ will tend to have opposite signs when vertices are sparsely connected, and the same sign when they are densely connected. Thus, to minimize $\lambda_2(R)$, it is good to put large weight on entries $r_{v,u}$ connecting the sparsely connected Fiedler cut associated with $\mathbf{x}$.

A *subgradient* of μ at R is a symmetric matrix J satisfying the following inequality for all $\tilde{R}$:

$$\mu(\widetilde{R}) \ge \mu(R) + \text{Trace}(J(\widetilde{R} - R))$$

This property guarantees that for small enough α, $\mu(R - \alpha J) < \mu(R)$. For our problem, we need to minimize $\mu(R)$ over the set of Markov Chains on $\mathcal{G}^*$, so our updates need to stay within this feasible set.

When $\mu(R) = \lambda_2(R)$ (a similar computation applies when $\mu(R) = -\lambda_n(R)$), the subgradient at R is $J = \mathbf{x}_2\mathbf{x}_2^T$, where $\mathbf{x}_2$ is the eigenvector associated with $\lambda_2(P)$. This can be seen by writing $\mu(R) = \mathbf{x}_2^T R\mathbf{x}_2$ and expressing $\mu(\widetilde{R})$ in terms of $\mathbf{x}_2$:

$$\begin{aligned}\mu(\widetilde{R}) &\ge \lambda_2(\widetilde{R})\\ &\ge \mathbf{x}_2^T\widetilde{R}\mathbf{x}_2\\ &= \mathbf{x}_2^T\widetilde{R}\mathbf{x}_2 + \mu(R) - \mathbf{x}_2^T R\mathbf{x}_2\\ &\ge \mu(R) - \mathbf{x}_2^T(\widetilde{R} - R)\mathbf{x}_2\\ &= \mu(R) - \sum_{i,j}\mathbf{x}_2[i]\mathbf{x}_2[j](\widetilde{r}_{i,j} - r_{i,j})\\ &= \mu(R) - \text{Trace}((\mathbf{x}_2^T\mathbf{x}_2)(\widetilde{R} - R)).\end{aligned}$$

A similar computation shows that when $\mu(R) = -\lambda_n$, $J = -\mathbf{x}_n^T\mathbf{x}_n$ is a subgradient. The subgradient method in each step moves from R to $R - \alpha J$, for some step size α. When $R - \alpha J$ is outside the feasible set $\{R : R \geq 0, R\mathbf{1} = \mathbf{1}\}$, it is projected back into the feasible set. The projection of minimum distance can be computed exactly by solving a quadratic program; however, for computational efficiency, we use an iterative projection method due to Boyd et al. (2004) described in Algorithm 13.

Algorithm 13: Compute the Fastest Mixing Markov Chain R

```
Result: Transition matrix $R$ of the Fastest Mixing Markov chain on $\mathcal{G}_r$ ;
Input: Graph $\mathcal{G}_r = (\mathcal{V}^*, \mathcal{E}_r)$ ;
Number of steps numIters ;
Initialize $R$ with the transition matrix of any symmetric Markov chain (e.g., Metropolis
  Hastings) on $\mathcal{G}_r$;
Let $\mathbf{p}$ denote a vector of the $r_{v,u}$ for $(v,u) \in \mathcal{E}_r$ ;
for $i \leftarrow 1$ to numIters do
    Compute sub-gradient: $\mathbf{j}^{(i)}$ of $\mathbf{p}$: $\mathbf{j}^{(i)}[(v,u)] = (\mathbf{x}_u - \mathbf{x}_v)^2$ ;
    Take Step: $\mathbf{p} = \mathbf{p} - \alpha_i \mathbf{j}^{(i)} / \parallel \mathbf{j}^{(i)} \parallel_2$ ;
    Project to positive orthant: ;
    for $v \in \mathcal{V}^*$, $\mathcal{I}(v) = \{e = (v,u)\ s.t.\ (v,u) \in \mathcal{E}_r\}$ do
        while $\sum_{e \in \mathcal{I}(v)} \mathbf{p}_e > 1$ do
            $\mathcal{I}(v) = \{e \text{ s.t. } e \in \mathcal{E}_r \text{ and } \mathbf{p}_e > 0\}$ ;
            $\delta = \min\{\min_{e \in \mathcal{I}(v)} \mathbf{p}_e, (\sum_{e \in \mathcal{I}(v)} \mathbf{p}_e - 1)/|\mathcal{I}(v)|\}$ ;
            $\mathbf{p}_e = \mathbf{p}_e - \delta$, for all $e \in \mathcal{I}(v)$ ;
        end
    end
end
Place remaining probability on self loops ;
for $v \in \mathcal{V}^*$ do
    $q_v = 0$ ;
    for $u \in \mathcal{V}^*$ s.t. $(v,u) \in \mathcal{E}_r$ do
        Increase $q_v$ by $\mathbf{p}_{(v,u)}$ ;
        $r_{v,u} = r_{v,u} = \mathbf{p}_{(v,u)}$;
    end
    $r_{v,v} = 1 - q_v$ ;
end
```

8.3.1 Combining the Standard Random Walk and Fastest Mixing Chain

While the FMMC does a better job of sampling edges across sparse cuts, one might be concerned that it might *oversample* such edges: for example, if the noise introduced by the GAN were to result in many other edges across a sparse cut, the oversampling of the existing few edges may result in losing the sparsity of the cut altogether. A second concern is that the FMMC may set $r_{v,u} = 0$ for edges $(v, u) \in \mathcal{E}^*$; then, the GAN would never see samples of these edges. To address both concerns, we consider walks that mix between the standard random walk and the FMMC.

For a given input graph $\mathcal{G}^*$, we denote the transition matrix of the FMMC with uniform stationary distribution by R, and the transition matrix of the standard random walk by Q. We consider the walk with transition matrix $\frac{1}{2}R + \frac{1}{2}Q$ which in each iteration flips a fair coin, and accordingly either takes a step according to the standard random walk or according to the FMMC. The walk $\frac{1}{2}R + \frac{1}{2}Q$ has some stationary distribution $\boldsymbol{\pi}$, and we choose the initial vertex v_1 according to π. Notice that this is in contrast to the standard NetGAN random walk algorithm, which chooses the initial vertex uniformly, even though the stationary node distribution of the node2vec walk is not uniform (Algorithm 10). We call the combination $\frac{1}{2}R + \frac{1}{2}Q$ the *combination walk* and do extensive experiments. We also tried other combinations on the Five Cluster graph and found the results to be as expected: when the coin is biased towards Q, the empirical edge probabilities are more uniform and when the bias is towards R, empirically edges across sparse cuts are seen more often.

8.3.2 Stationary distributions over edges

In order to compare the probability distributions for visiting edges induced by (1) the node2vec Markov chain, (2) the Fastest Mixing Markov Chain R, and (3) the combination walk $\frac{1}{2}R + \frac{1}{2}Q$, we compute, for each chain, the stationary distribution using edges as states. While the stationary distributions for R and Q are easily computed in the standard way using vertices as states, the second-order nature of the node2vec walk makes it more amenable to analysis as a random walk on *edges*. The corresponding transition matrix R has entries R_{e_1,e_2} equal to the probability of transitioning from $e_1 = (u_1, v_1)$ to $e_2 = (u_2, v_2)$. The edges e_1, e_2 are treated as directed. If $v_1 \neq u_2$, then $R_{e_1,e_2} = 0$, and $R_{e_1,e_2} = 1/d_{v_1}$ when $v_1 = u_2$. The stationary distribution $\boldsymbol{\pi}$ is over directed edges. Note that for the first order chain, $R_{f,e}$ is independent of f_1.

The stationary edge distributions for each chain on the Five Cluster graph are shown in Figure 8.1. Recall that the standard random walk has probability $1/(2m)$ for each edge. The stationary distributions of all node2vec walks resembles normal distributions sharply concentrated around $\frac{1}{2m^*}$; this is not very surprising, given that the intention of the z_1, z_2 parameters is not to alter the stationary probabilities, but to change the DFS/BFS dynamics of the walk. It might also explain why there is empirically little differentiation between graph generators with different settings of z_1, z_2. Unlike the node2vec walks, the tail for the stationary distributions of R and $\frac{1}{2}R + \frac{1}{2}Q$ is quite long (notice the different scales of the axes), with a maximum value at 0.00047 and a mean at 0.000046. The high edge probabilities correspond to inter-cluster edges, as expected.

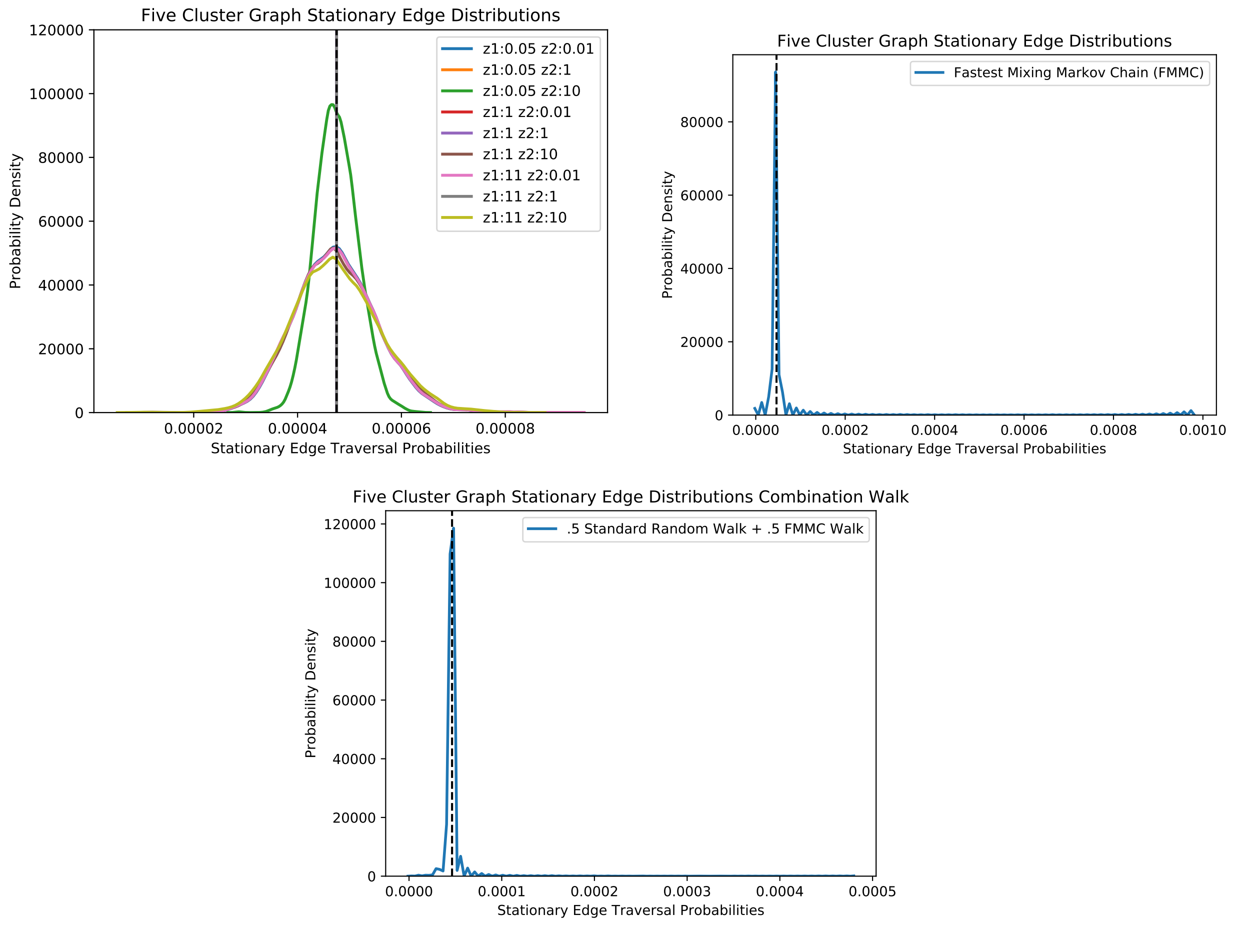


Figure 8.1: Stationary distributions over directed edges for the node2vec walk, Combination walk, and Fastest Mixing Markov Chain walk.

8.4 Memory and lengths of walks

By reducing $\mathcal{W}(\hat{\boldsymbol{\theta}}_g)$ to just the edge frequency matrix $\tilde{A}$, any structural information about the walk, i.e., the sequence of edges, the length, is discarded. Thus, at stationarity, up to normalization, the matrix $\tilde{A}$ will be the same regardless of the length of the sampled walks. In light of this observation, the role played by the length of the walks is investigated in Section 8.9.3. The exact same output frequencies would be obtained if only walks of a single edge were sampled, so what is the role of longer walks in the learning process?

One answer is that if walks are short (in the extreme case: only a single edge), the discriminator has to essentially memorize all of $\mathcal{G}^*$ to detect fake walks. Then, even if the generator had the capacity to learn a good distribution, it would not do so because it would be missing the feedback from a sufficiently powerful discriminator. Longer walks give the discriminator more opportunities to detect invalid steps or possibly "jumps" between nodes, thus forcing the generator to generate walks from a more realistic distribution.

The structural information lost in the reduction of $\mathcal{W}(\hat{\boldsymbol{\theta}}_g)$ to $\tilde{A}$ also raises questions about the role of memory in the generator RNN. If the true distribution is first-order Markovian (like the standard random walk, the combination walk, and the FMMC), then the only capabilities that a generator should need to sample from these distributions are: (1) converting initial Gaussian noise to a start vertex sampled (approximately) from the stationary distribution of the walk, and (2) given a current vertex $\boldsymbol{w}[t]$, sample the next vertex $\boldsymbol{w}[t+1]$ (approximately) from the correct transition probabilities. Since the process is Markovian, no information besides $\boldsymbol{w}[t]$ should be needed. Then what is the role of memory states for the generator and discriminator?

In Section 8.9.3, we report on experiments which change the memory states used by LSTM. Two of the extremes are shortening random walks to have length 2, and setting all memory states $\mathbf{m}_t = 0$ for the generator, effectively forcing the walk distribution to be truly Markovian.

8.5 Edge-Independent Sampling

Fixed-edge iterative sampling is designed to make disconnected graphs less likely. We are interested in seeing if using FMMC alone is sufficient to achieve this goal and whether we can use a simpler graph sampling technique instead. The *edge-independent sampling (EI)* approach includes each edge (v, u) independently with probability $a_{v,u} \approx \min(1, m^* \cdot \tilde{a}_{v,u}/Z)$ (where $Z = \sum_{v,u} \tilde{a}_{v,u}$ is a normalizing constant). To push the entry sum closer to m, we compute the difference $\delta = m - \sum_{v,u} a_{v,u}$. If positive and the set $X = \{(v, u) \text{ s.t. } a_{v,u} \in (0, 1)\}$ is non-empty, we let $Z = \sum_{(v,u)\in X} a_{v,u}$ and

Algorithm 14: ScaleMatrix

```
Result: Probabilistic adjacency matrix $A$
Input: Frequency matrix $\tilde{A}$;
Number of edges $m^*$;
Normalization factor $Z = \sum_{v,u} \tilde{a}_{v,u}$ ;
$a_{v,u,=} \min(1, m^* \cdot \tilde{a}_{v,u}/Z)$ ;
$\delta = m - \sum_{v,u} a_{v,u}$; while $\delta > \epsilon$ do
    $X = \{(v,u) | a_{v,u} \in (0,1)\}$ ;
    If $X = \emptyset$, break ;
    Else, $Z = \sum_{(v,u) \in X} \tilde{a}_{v,u}$ ;
    $a_{v,u} = \min(1, m^* \cdot a_{v,u}/Z)$ for all $(v,u) \in X$ ;
end
```

$a_{v,u} = \min(1, m'/Z) \cdot a_{v,u}$ where $m' = m - |\{(v,u) s.t. a_{v,u} = 1\}|$. The procedure is terminated once the sum reaches m or all positive entries are 1.

After constructing A, $\mathcal{G}$ is sampled using randomized rounding (Definition 2.3.1).

8.6 Early-Stopping

While it is important for all learning tasks to avoid overfitting, the issue is perhaps even more acute in the NetGAN framework. That is because the training data is generated from a single graph, yet we are interested in generating diverse graphs. Meeting the diversity criterion when training with random walks is non-trivial because the generator g is trained using walks that contain only edges in $\mathcal{E}^*$. Thus, if g learns the walks' edge distribution too well, such as by memorizing $\mathcal{E}^*$, it can only produce subgraphs of $\mathcal{G}^*$. To obtain more diversity, learning must be curtailed sufficiently that g still produces walks including edges not in $\mathcal{E}^*$.

As noted above in Section 8.1.4, if there is a perfect threshold classifier then edge classification is perfect measured by ROC AUC and AP. While bad edge prediction is a clear indication of overfitting, good edge prediction under the ROC AUC and AP scores is not necessarily indicative of generating similar graphs, mostly because even small differences between frequencies of edges and non-edges are enough to obtain a good threshold classifier, but also because structurally important edges are treated no differently than other edges.

We extend the stopping criterion based on ROC AUC and AP used by Bojchevski et al. (2018) (Algorithm 11). Bojchevski et al. (2018) stop training once there has been z consecutive phases of ℓ training iterations where the sum of the ROC AUC score and AP score has not grown. We call this criterion the *Edge Prediction (EP) criterion* because it measures how well $\tilde{A}$ performs at predicting edges measured by ROC AUC and AP. Our stopping criterion uses ROC AUC and AP and uses the expected spectral gap of the generated graph measured by the spectrum of $L(A)$. Chung and Radcliffe (2011) shows that when a graph $\mathcal{G}$ with adjacency matrix B is generated using randomized-rounding on A, the spectral gap of $L(B)$ is sharply concentrated around that of $L(A)$ (Theorem 10). Hence, the spectral gap of $L(A)$ is a good approximation for that of the graph generated from A using randomized-rounding. We combine the stopping criteria by computing the ROC AUC and AP of $\tilde{A}$ and the spectral gap of A every ℓ training iterations. The optimization criteria are to maximize the ROC AUC and AP, and to minimize $|\lambda_2(L(A^*)) - \lambda_2(L(A))|$. Learning is terminated once for each of the criteria, there have at least once been z consecutive phases of ℓ iterations each when that criterion did not improve (Algorithm 15). We call this stopping criterion the *spectral stopping (SP+EP) criterion.*

8.7 Experimental results

In Sections 8.8 and 8.9 we discuss our experimental results. We conduct experiments on the Football, Netscience, Five Cluster, Airport, and Email graphs all discussed in Chapter 6.

Throughout we refer to the graph features used to compare the simlarity of graphs and the diversity metrics using abbreviations and symbols, Table 8.2 is a reference to these abbreviations.

8.8 What is the NetGAN learning?

8.8.1 Tracking Progress

To better compare the progress of learning the true random walk distribution during GAN training, we introduce two distributions over edges: (1) the probability of traversing an edge (v, u) during a random walk with transition matrix T at stationarity $z^{(T)}$ equal to $\pi_v T_{v,u}$ (2) distribution $y^{(\tilde{A})}$ with $y^{(\tilde{A})}(u, v) = \tilde{a}_{u,v}/Z$. We track $d_{\mathrm{TV}}(z^{(T)}, y^{(\tilde{A})})$ as a proxy for how close the edge distribution learned by the NetGAN is to the edge distribution induced by walk transition matrix T[6].

[6]Here we use Total-Variation distance instead of Earth Mover's distance because there is not a clear metric to apply on the set of all node pairs.

Algorithm 15: Spectrum and Edge Prediction Stopping Criterion

```
Result: Yes if all three stopping measures (AP, AUC, and SpecGap) have had $z$ subsequent
        evaluations where it did not improve. Else, No.
Input: Positive validation edges $\mathcal{E}_v$;
Negative validation edges $\tilde{\mathcal{E}_v}$;
Current frequency matrix $\tilde{B}$;
Most recent frequency matrix that had the best AP, AUC, or SpecGap $\tilde{A}$;
Spectral gap of input matrix $\lambda_2(A^*)$ ;
*Best scores found for each evaluation criterion:* bestAuc, bestAp, bestSpecGapDiff ;
*Number of evaluations remaining for each evaluation criterion before termination:* ;
evalsLeftAuc, evalsLeftAp, evalsLeftSpecGap ;
*Number of evaluations tolerated without improving: z*
True scores $\mathbf{x} = \{1 for(u,v) \in \mathcal{E}_v, 0 for(u,v) \in \tilde{\mathcal{E}_v}\}$ ;
Valid scores $\mathbf{y} = \{\tilde{a}_{\prime,u} v for(u,v) \in \mathcal{E}_v \cup \tilde{\mathcal{E}_v}\}$ ;
$B$ is ScaleMatrix($\tilde{B}$) (Algorithm 14) ;
if $evalsLeftAuc > 0$ then
    If AUC($\mathbf{x},\mathbf{y}$) $\leq$ bestAuc, evalsLeftAuc = evalsLeftAuc $- 1$;
    Else, bestAuc = AUC($\mathbf{x},\mathbf{y}$), evalsLeftAuc = $z$ and $\tilde{A} = \tilde{A}'$ ;
if $evalsLeftAp > 0$ then
    If AP($\mathbf{x},\mathbf{y}$) $\leq$ bestAp, evalsLeftAp = evalsLeftAp $- 1$;
    Else, bestAp = AP($\mathbf{x},\mathbf{y}$), evalsLeftAp = $z$ and $\tilde{A} = \tilde{A}'$ ;
if $evalsLeftSpecGap > 0$ then
    If $|\lambda_2(A') - \lambda_2(A^*)| \geq$ bestSpecGapDiff, evalsLeftSpecGap = evalsLeftSpecGap $- 1$;
    Else, bestSpecGapDiff = $|\lambda_2(A') - \lambda_2(A^*)|$, evalsSpecGap = $z$ and $\tilde{A} = \tilde{A}'$ ;
if $evalsLeftAp = evalsLeftSpecGap = evalsLeftAuc = 0$ then
    Return Yes
else
    Return No
end
```

Full name	Abbreviated name
Fraction of probability mass in a probabilistic graph that is on $\mathcal{E}^*$ (Edge overlap)	EO
Size of largest connected component of graph	Size LCC
Earth Mover's Distance	EMD
Clustering Coefficient	CC
Shortest-path Distance	SP
Betweenness centrality	BTWN
Degree	DEG

Table 8.2: Table of the abbreviated names of graph similarity and diversity metrics.

8.8.2 Tracking mass on non-edges

We discussed earlier that the GAN's role appears to be to introduce controlled noise into the edge frequency matrix. One trivial way to add noise — which would not even require a GAN — would be to produce an existing random (uniform or otherwise) edge with probability p, and a uniformly random non-existing pair (u, v) with probability $1 - p$. Here, we discuss experiments that test whether the GAN learns a more "interesting" noise distribution.

We test how uniform the noise is over the set of "noise pairs" $(u, v) \notin \mathcal{E}^*$. We call mass on "noise pairs" addition noise (in contrast to subtraction noise which would be omitting traversals over pairs $(u, v) \in \mathcal{E}^*$). We compute the average $\overline{x} = (\sum_{(u,v)\notin\mathcal{E}^*} a_{u,v})/(n^2 - |\mathcal{E}^*|)$ to measure the average addition noise. If the addition noise was applied uniformly, all noise pairs would have $a_{u,v} = \overline{x}$. For both walks, once $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(T)}) \leq .6$, the average deviation $|a_{u,v} - \overline{m}|$ is at least $\overline{x}$; for the NETSCIENCE, AIRPORT, and EMAIL graphs, this holds as early as $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(T)}) \leq .9$ (Figure 8.2). We also see that for both walks, the average addition noise decreases with $d_{\mathrm{TV}}(z^{(T)}, y^{(\tilde{A})})$.

One source of this deviation of addition noise from uniform is that the generator trained with either walk adds less noise across the *Fiedler cut* (Definition 5.0.4) than there would be if noise were distributed uniformly (Figure 8.3). More noise is added across the Fiedler cut for the combination walk than the standard walk, suggesting that training with walk distributions that place more weight on edges crossing sparse cuts results in the generator more frequently sampling not only those high-weight edges but noise pairs crossing the sparse cuts as well.

8.9 Comparing NetGAN Variants

8.9.1 Statistics on A across NetGAN variants

First, we compare the properties of A across different walks and stopping criteria across five of our datasets (Table 8.3). For our datasets, we observe that for both stopping methods that the standard walk out performs the combination walk on matching the spectrum using the ℓ_2^{LW} metric. We also observe that this gain in matching the spectrum typically comes with a decrease in entropy. We also notice that for both stopping criterion that $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(T)})$ is smaller for the standard walk than the combination walk, which is consistent with that fact that the standard walk matches $\boldsymbol{\lambda}^*$

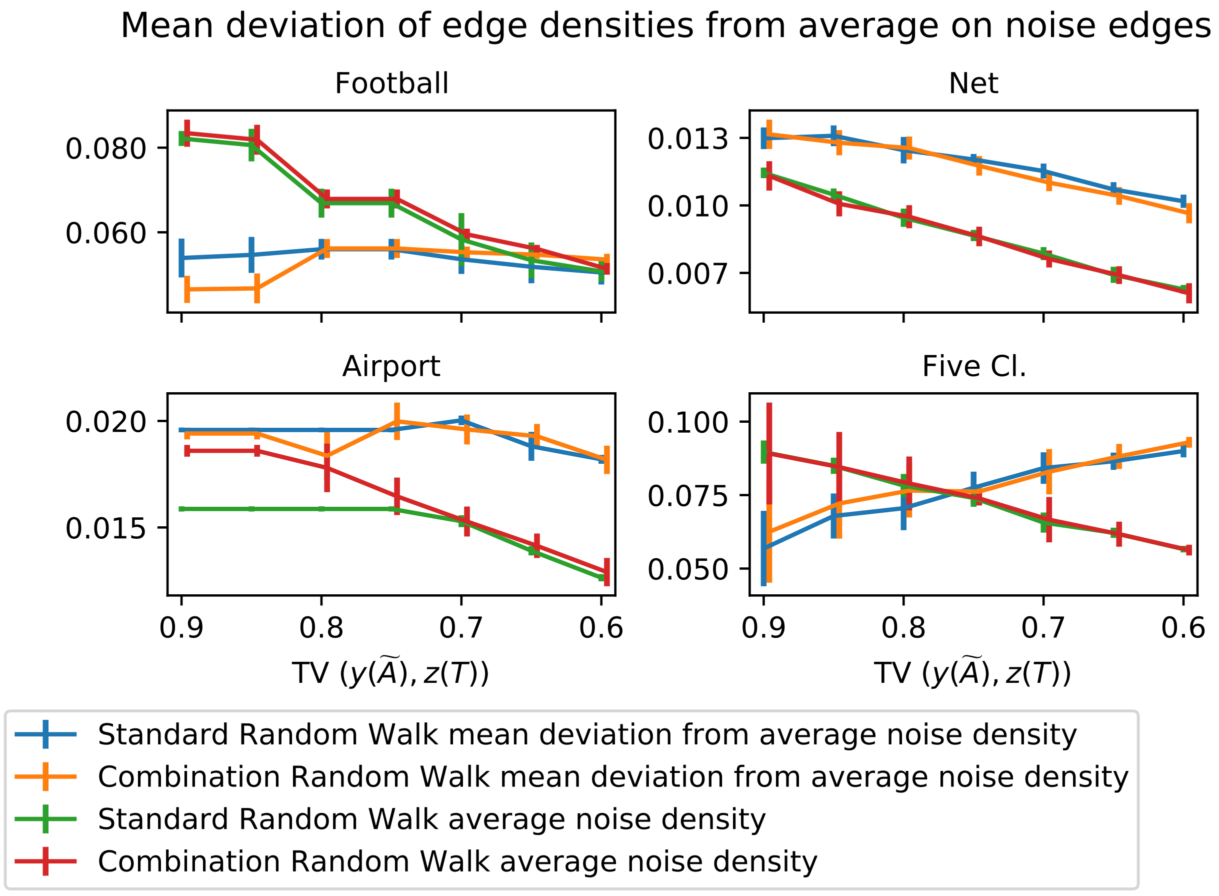


Figure 8.2: Average deviation of addition noise from uniform against the Total-Variation distance between the edge densities learned by the NetGAN and the random walk edge traversal probabilities.

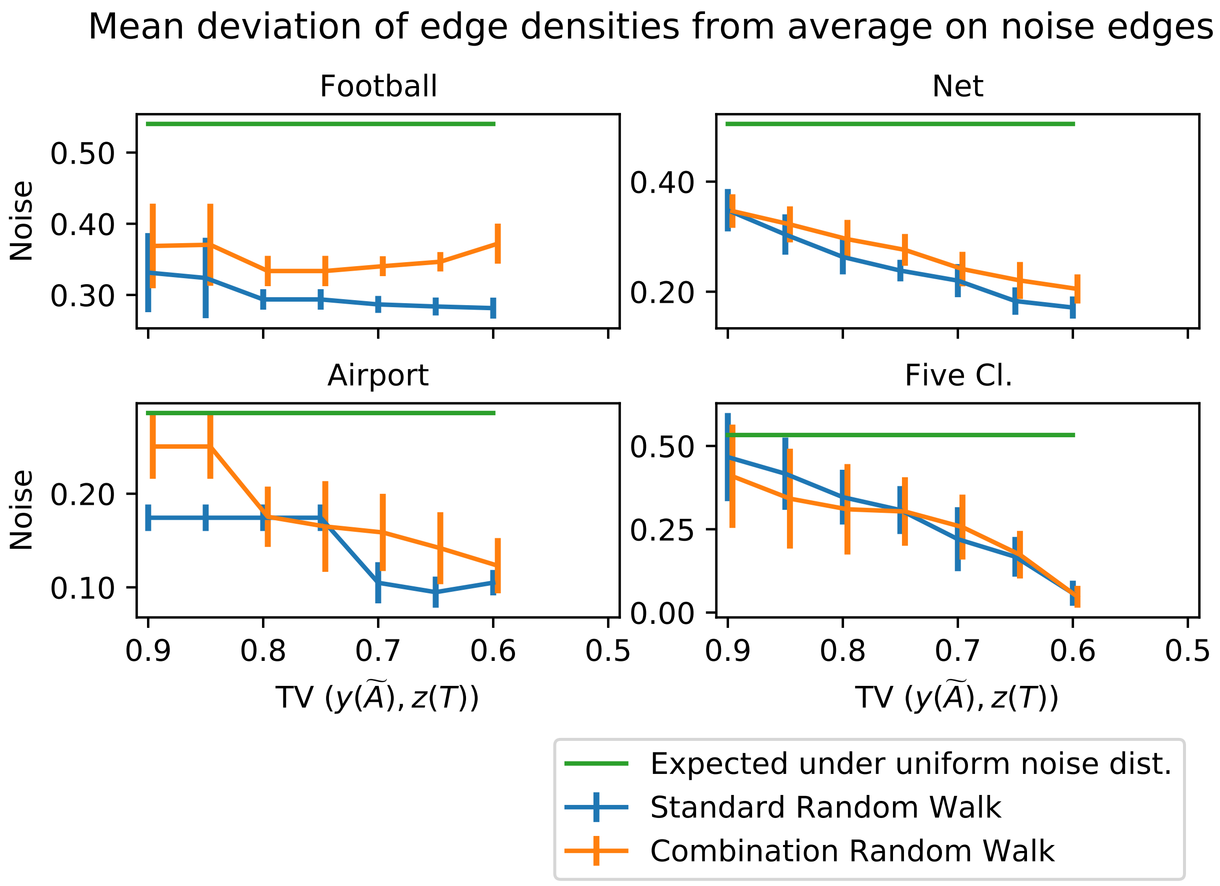


Figure 8.3: Addition noise across Fiedler cut against the Total-Variation distance between the edge densities learned by the NetGAN and the random walk edge traversal probabilities.

better because it is closer to a uniform distribution over $\mathcal{E}^*$.

Table 8.3: Properties of probababilistic adjacency matrix A for different NetGAN walks, data sets, and termination criteria.

FOOTBALL

WALK/STOP	$d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(T)})$	$\lvert\lambda_2^* - \lambda_2\rvert$	ℓ_2^{LW} SPEC.	ROC AUC	AP	EO	H(A)
STD./EP	0.51 ± 0.06	0.08 ± 0.02	0.11 ± 0.07	0.88 ± 0.03	0.9 ± 0.03	0.55 ± 0.07	0.17 ± 0.02
STD./SPEC+EP	0.42 ± 0.08	0.05 ± 0.02	0.05 ± 0.02	0.87 ± 0.03	0.91 ± 0.02	0.67 ± 0.12	0.13 ± 0.04
COMBO./EP	0.67 ± 0.05	0.19 ± 0.02	0.26 ± 0.08	0.86 ± 0.06	0.89 ± 0.04	0.39 ± 0.06	0.22 ± 0.02
COMBO./SPEC+EP	0.19 ± 0.02	0.05 ± 0.01	0.01 ± 0.0	0.78 ± 0.04	0.78 ± 0.06	0.98 ± 0.01	0.04 ± 0.0

NETSCIENCE

WALK/STOP	$d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(W)})$	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} SPEC.	ROC AUC	AP	EO	H(A))
STD./EP	0.39 ± 0.1	0.05 ± 0.02	0.27 ± 0.29	0.98 ± 0.01	0.98 ± 0.01	0.69 ± 0.11	0.02 ± 0.01
STD./SPEC+EP	0.26 ± 0.05	0.03 ± 0.01	0.07 ± 0.04	0.98 ± 0.02	0.98 ± 0.02	0.84 ± 0.06	0.01 ± 0.0
COMBO./EP	0.4 ± 0.09	0.04 ± 0.02	0.15 ± 0.11	0.98 ± 0.02	0.98 ± 0.02	0.7 ± 0.1	0.02 ± 0.01
COMBO./SPEC+EP	0.31 ± 0.04	0.02 ± 0.01	0.04 ± 0.02	0.99 ± 0.01	0.99 ± 0.01	0.81 ± 0.07	0.01 ± 0.0

FIVE CLUSTER

WALK/STOP	$d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(W)})$	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} SPEC.	ROC AUC	AP	EO	H(A))
STD./EP	0.57 ± 0.01	0.0 ± 0.01	0.08 ± 0.01	0.94 ± 0.01	0.9 ± 0.01	0.5 ± 0.01	0.14 ± 0.01
STD./SPEC+EP	0.56 ± 0.0	0.0 ± 0.0	0.08 ± 0.0	0.94 ± 0.01	0.9 ± 0.02	0.51 ± 0.01	0.13 ± 0.0
COMBO./EP	0.58 ± 0.02	0.01 ± 0.02	0.09 ± 0.02	0.94 ± 0.01	0.9 ± 0.01	0.49 ± 0.02	0.15 ± 0.02
COMBO./SPEC+EP	0.57 ± 0.01	0.0 ± 0.0	0.07 ± 0.01	0.94 ± 0.01	0.9 ± 0.01	0.51 ± 0.01	0.13 ± 0.0

AIRPORT

WALK/STOP	$d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(W)})$	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} SPEC.	ROC AUC	AP	EO	H(A))
STD./EP	0.45 ± 0.05	0.2 ± 0.08	0.81 ± 0.29	0.98 ± 0.01	0.98 ± 0.01	0.67 ± 0.05	0.04 ± 0.0
STD./SPEC+EP	0.39 ± 0.06	0.09 ± 0.07	0.41 ± 0.27	0.98 ± 0.01	0.98 ± 0.01	0.75 ± 0.06	0.03 ± 0.01
COMBO./EP	0.71 ± 0.06	0.18 ± 0.08	1.17 ± 0.39	0.97 ± 0.01	0.97 ± 0.01	0.37 ± 0.08	0.07 ± 0.01
COMBO./SPEC+EP	0.62 ± 0.12	0.11 ± 0.06	0.69 ± 0.46	0.97 ± 0.01	0.97 ± 0.01	0.49 ± 0.15	0.06 ± 0.01

EMAIL

WALK/STOP	$d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(W)})$	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} SPEC.	ROC AUC	AP	EO	H(A))
STD./EP	0.79 ± 0.04	0.11 ± 0.02	0.74 ± 0.17	0.92 ± 0.01	0.92 ± 0.01	0.23 ± 0.04	0.03 ± 0.0
STD./SPEC+EP	0.63 ± 0.14	0.07 ± 0.04	0.39 ± 0.28	0.91 ± 0.01	0.92 ± 0.01	0.42 ± 0.16	0.02 ± 0.01
COMBO./EP	0.84 ± 0.04	0.12 ± 0.02	0.66 ± 0.22	0.91 ± 0.02	0.91 ± 0.02	0.19 ± 0.05	0.03 ± 0.0
COMBO./SPEC+EP	0.76 ± 0.09	0.09 ± 0.03	0.4 ± 0.19	0.9 ± 0.02	0.91 ± 0.02	0.28 ± 0.1	0.03 ± 0.0

To better compare the properties of A when training with the two walks, we compare performance for both walks once training has reached equal $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(T)})$ in Figure 8.4. Controlling for $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(T)})$, we observe the combination walk at least matches or has smaller/larger ℓ_2^{LW}/entropy.

We see that the SP+EP stopping criterion was typically met after EP, sometimes as much as 10000 training iterations later. In our experiments, when SP+EP was met significantly after EP, we see a decrease in ROC AUC, AP, $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(T)})$, and average entropy on the entries of A accompanied by a huge increase in EO (we see this most when training with combination walk on the FOOTBALL graph). This indicates overfitting. In terms of the spectrum itself, the additional training iterations not only improved the expected spectral gap, but typically led to drops in the ℓ_2^{LW} objective as well.

To illustrate the effects of the spectrum stopping criterion in more detail, we visualize the adjacency matrices of the FIVE CLUSTER graph where the known graph structure makes a visual representation of the effects easier. On average, the spectrum criterion was met 2000 iterations after the edge prediction criterion. Figure 8.5 shows a typical heat map of A after 4500 iterations (when the prediction criterion was met) and after 6500 iterations (when the spectrum criterion was met). At both points in time, higher scores are assigned to intra-cluster edges than inter-cluster ones; however, after 4500 iterations, the density of inter-cluster edges is still approximately .1 when it should only be approximately .0001. Most of these excessive edges between clusters disappear when the spectrum criterion is met.

8.9.2 Discrete graph sampling and matching discrete graph properties

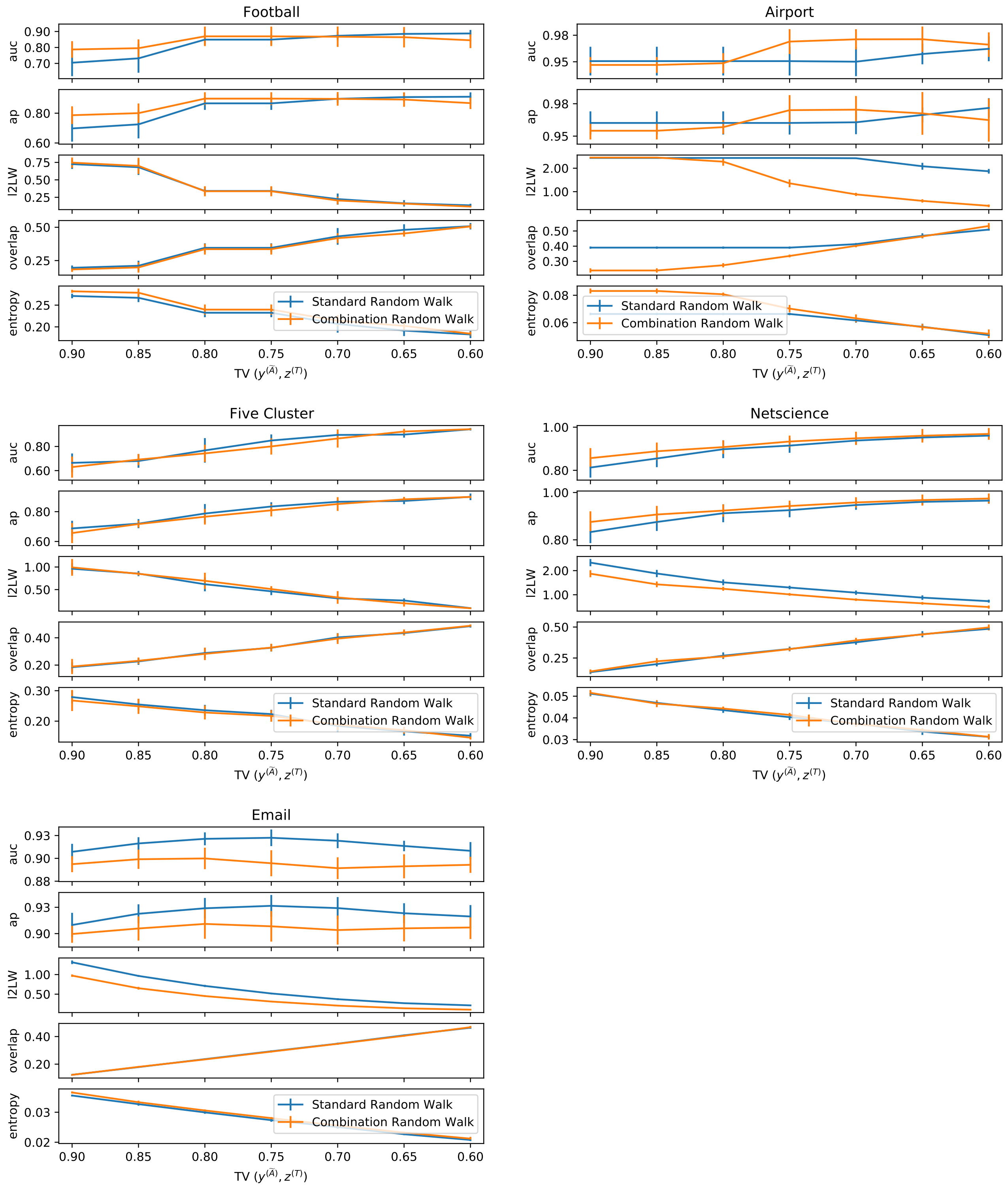


Figure 8.4: Statistics of probababilistic adjacency matrix A against the Total-Variation distance between the edge densities learned by the NetGAN and the random walk edge traversal probabilities.

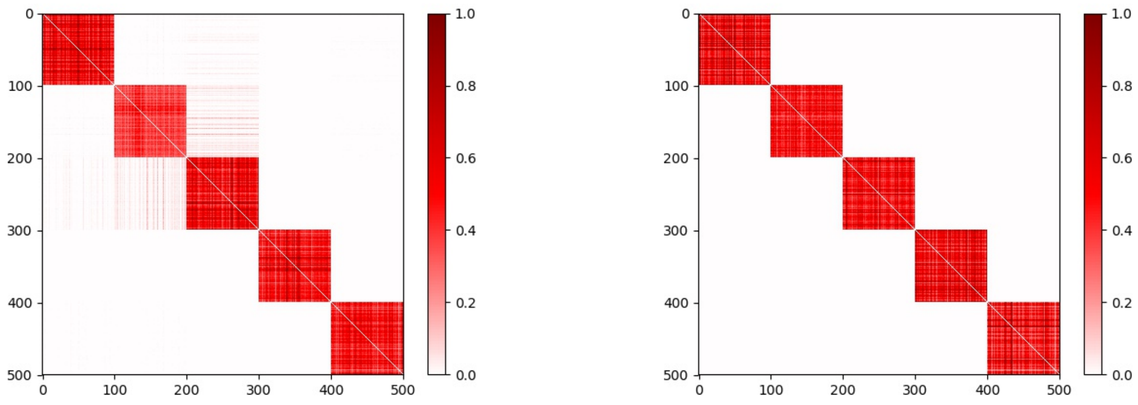


Figure 8.5: Heat maps for edge densities learned by NetGAN on the Five Cluster graph. Longer training improves the estimates of density of inter-cluster edges in the Five Cluster graph.

Table 8.4: Properties of largest connected component of graphs drawn with edge independent sampling and SPEC+EP NetGAN stopping criteria.

			Football				
Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	**0.02±0.0**	**0.01±0.0**	115±0	0.04±0.0	**0.06±0.0**	0.0±0.0	1.66±0.2
Combo.	0.04±0.0	0.01±0.0	**115±0**	0.04±0.0	0.08±0.0	**0.0±0.0**	**1.05±0.1**
			Netscience				
Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	0.01±0.0	0.02±0.0	**367±9**	1.26±0.2	0.25±0.0	**0.01±0.0**	**0.27±0.1**
Combo.	**0.01±0.0**	**0.01±0.0**	362±9	**1.05±0.3**	**0.23±0.0**	0.01±0.0	0.49±0.1
			Five Cluster				
Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	0.0±0.0	0.11±0.1	428±83	**2.02±0.4**	0.05±0.0	**0.01±0.0**	5.94±0.5
Combo.	**0.0±0.0**	**0.01±0.0**	**497±7**	2.5±0.4	**0.04±0.0**	0.01±0.0	**5.47±0.6**
			Airport				
Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	**0.05±0.0**	**0.09±0.1**	476±18	0.18±0.1	**0.17±0.0**	**0.0±0.0**	**1.15±0.4**
Combo.	0.08±0.0	0.1±0.1	**479±11**	**0.12±0.0**	0.34±0.1	0.0±0.0	4.2±1.1
			Email				
Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	**0.03±0.0**	0.02±0.0	1068±26	0.2±0.1	**0.06±0.0**	**0.0±0.0**	**0.65±0.2**
Combo.	0.04±0.0	**0.02±0.0**	**1117±6**	**0.13±0.0**	0.11±0.0	0.0±0.0	1.84±0.2

Table 8.5: Properties of largest connected component of graphs drawn with edge independent sampling and EP NetGAN stopping criteria.

FOOTBALL

WALK	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} SPEC.	SIZE LCC	SP. EMD	CC. EMD	BTWN. EMD	DEG. EMD
STD.	**0.04±0.0**	**0.01±0.0**	115±0	**0.06±0.0**	**0.08±0.0**	0.0±0.0	1.84±0.1
COMBO.	0.12±0.0	0.06±0.0	**115±0**	0.15±0.0	0.2±0.0	**0.0±0.0**	**1.83±0.1**

NETSCIENCE

WALK	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} SPEC.	SIZE LCC	SP. EMD	CC. EMD	BTWN. EMD	DEG. EMD
STD.	0.02±0.0	0.04±0.0	**355±9**	1.48±0.3	0.32±0.1	**0.01±0.0**	**0.44±0.2**
COMBO.	**0.02±0.0**	**0.02±0.0**	354±10	**1.23±0.3**	**0.3±0.1**	0.01±0.0	0.65±0.1

FIVE CLUSTER

WALK	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} SPEC.	SIZE LCC	SP. EMD	CC. EMD	BTWN. EMD	DEG. EMD
STD.	**0.0±0.0**	0.03±0.0	483±33	**2.53±0.4**	**0.05±0.0**	**0.01±0.0**	**5.28±1.4**
COMBO.	0.01±0.0	**0.01±0.0**	**497±8**	2.83±0.4	0.07±0.0	0.01±0.0	5.68±1.3

AIRPORT

WALK	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} SPEC.	SIZE LCC	SP. EMD	CC. EMD	BTWN. EMD	DEG. EMD
STD.	**0.12±0.1**	**0.13±0.1**	462±15	**0.25±0.1**	**0.21±0.0**	**0.0±0.0**	**1.36±0.3**
COMBO.	0.13±0.1	0.16±0.0	**482±5**	0.15±0.0	0.41±0.0	0.0±0.0	4.73±0.4

EMAIL

WALK	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} SPEC.	SIZE LCC	SP. EMD	CC. EMD	BTWN. EMD	DEG. EMD
STD.	**0.05±0.0**	0.04±0.0	1045±12	0.26±0.0	**0.12±0.0**	**0.0±0.0**	**0.82±0.1**
COMBO.	0.06±0.0	**0.03±0.0**	**1112±3**	**0.19±0.0**	0.14±0.0	0.0±0.0	1.82±0.1

Table 8.6: Properties of largest connected component of graphs drawn with NetGAN fixed-edge sampling and SPEC+EP stopping criteria.

			Football				
Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	0.05±0.0	0.02±0.0	115±0	0.08±0.0	0.11±0.0	0.0±0.0	1.74±0.2
Combo.	**0.04±0.0**	**0.01±0.0**	**115±0**	**0.05±0.0**	**0.09±0.0**	**0.0±0.0**	**1.1±0.1**
			Netscience				
Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	0.02±0.0	0.02±0.0	**377±2**	1.32±0.1	0.33±0.0	0.01±0.0	0.23±0.0
Combo.	**0.02±0.0**	**0.01±0.0**	375±2	**1.15±0.3**	**0.32±0.1**	**0.01±0.0**	**0.44±0.1**
			Five Cluster				
Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	**0.0±0.0**	0.08±0.1	446±61	**2.06±0.3**	0.03±0.0	0.01±0.0	3.35±0.3
Combo.	0.0±0.0	**0.0±0.0**	**498±6**	2.62±0.3	**0.02±0.0**	0.01±0.0	**3.09±0.4**
			Airport				
Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	**0.07±0.1**	**0.09±0.1**	**500±1**	0.18±0.1	**0.24±0.0**	**0.0±0.0**	**1.56±0.3**
Combo.	0.09±0.0	0.13±0.1	499±1	**0.14±0.0**	0.4±0.1	0.0±0.0	4.81±1.1
			Email				
Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	**0.04±0.0**	**0.02±0.0**	1132±4	0.17±0.0	**0.1±0.0**	**0.0±0.0**	**0.56±0.1**
Combo.	0.06±0.0	0.03±0.0	**1133±0**	**0.15±0.0**	0.14±0.0	0.0±0.0	2.2±0.2

Table 8.7: Properties of largest connected component of graphs drawn with NetGAN fixed-edge sampling and EP stopping criteria.

Football

Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	**0.07±0.0**	**0.03±0.0**	115±0	**0.11±0.0**	**0.15±0.0**	0.0±0.0	**1.79±0.1**
Combo.	0.14±0.0	0.07±0.0	115±0	0.18±0.0	0.23±0.0	**0.0±0.0**	1.78±0.1

Netscience

Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	0.03±0.0	0.03±0.0	**378±0**	1.44±0.2	0.43±0.1	**0.01±0.0**	**0.3±0.0**
Combo.	**0.03±0.0**	**0.02±0.0**	377±2	**1.29±0.2**	**0.42±0.1**	0.01±0.0	0.54±0.1

Five Cluster

Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	**0.01±0.0**	0.03±0.0	486±29	**2.63±0.4**	**0.04±0.0**	0.01±0.0	**3.11±1.0**
Combo.	0.02±0.0	**0.02±0.0**	**499±4**	2.92±0.4	0.06±0.1	0.01±0.0	3.46±0.9

Airport

Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	**0.14±0.1**	**0.15±0.1**	**500±0**	0.25±0.0	**0.29±0.0**	**0.0±0.0**	**1.79±0.1**
Combo.	0.15±0.1	0.2±0.1	500±0	**0.17±0.0**	0.47±0.0	0.0±0.0	5.49±0.3

Email

Walk	$\lvert\lambda_2 - \lambda_2^*\rvert$	ℓ_2^{LW} Spec.	Size LCC	SP. EMD	CC. EMD	Btwn. EMD	Deg. EMD
Std.	**0.06±0.0**	**0.04±0.0**	1132±4	0.2±0.0	**0.15±0.0**	**0.0±0.0**	**0.59±0.1**
Combo.	0.08±0.0	0.04±0.0	**1133±0**	**0.19±0.0**	0.16±0.0	0.0±0.0	2.2±0.1

Tables 8.4, 8.5, 8.6 and 8.7 show statistics for the largest connected component of the discrete graphs generated. Training with combination walks on average generates larger connected components, more similar to the input graph's size; this discrepancy is largest for edge-independent graph generation on the Five Cluster graph. The combination walk tends to do worse than the standard walk for the remaining graph characteristics under the EP and SP+EP stopping criteria. One reason for this performance gap is that the number of training iterations required to meet the criterion for the standard walk exceeds the number for the combination walk, so the stopping criterion allows the standard walk to train for more iterations.

The differences in the graphs produced by fixed-edge and edge independent sampling are minor, with the fixed-edge sampling helping keep more nodes together but in general lagging slightly behind in all other metrics.

8.9.3 Walks of different lengths and role of memory

In our experiments, training with length-2 walks or with all memory states set to zero was unstable on the Five Cluster graph, and the approach did not produce any useful output graphs. For the Airport graph, the results were almost competitive, though not quite as good as for walks of length $k = 16$ with memory; the spectrum (and AUC and AP performance) is shown in Figure 8.6. We note that normalizing for the number of edges seen by the generator made a significant difference in the quality of training.

Figure 8.7 illustrates the failure to learn on the Five Cluster graph in more detail. For Five Cluster, a GAN using walks of length 2, or no memory states, was unable to learn. When training on length-2 walks, the loss at the beginning of the training goes to zero, meaning that the discriminator does not learn to tell fake walks from real even at the beginning, when the generator's samples are poor. This leads us to believe that the role of longer walks is predominantly to aid the discriminator[7].

Without memory, we observe that the training is extremely slow. We suspect that without memory, the gradients are vanishing, which is a known problem in Recurrent Neural Networks Hochreiter and Schmidhuber (1997), and one of the reasons that the memory state was introduced

[7]There could be alternative explanations, such as not training the discriminator for enough iterations before updating the generator, or not tuning the learning rate properly. We conducted a grid search for both of these possibilities, and performance did not improve, suggesting a more fundamental issue.

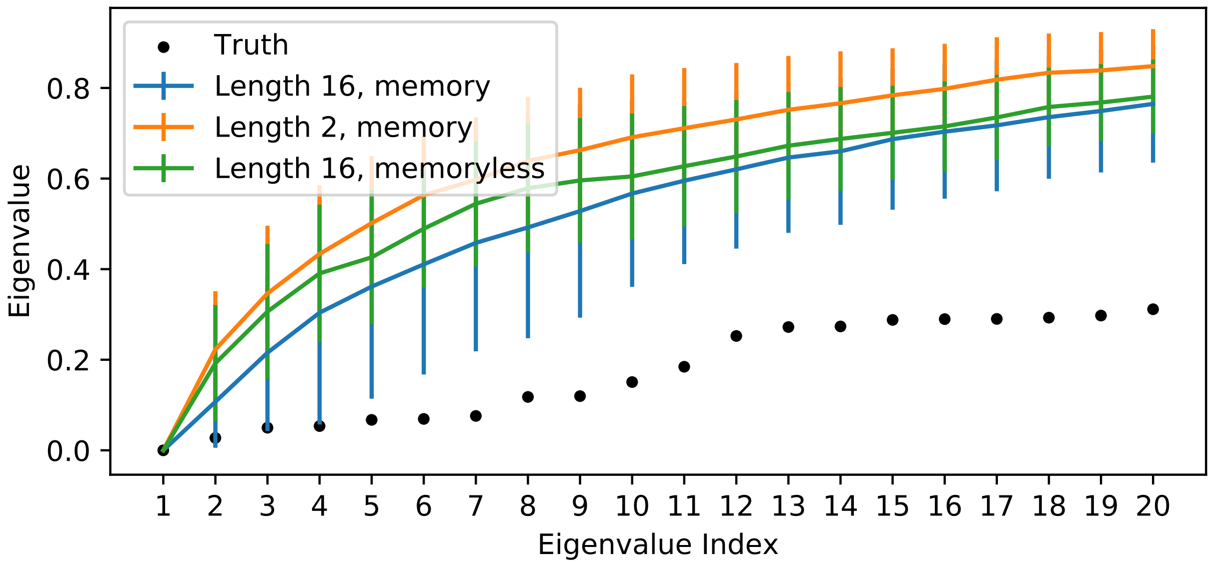


Figure 8.6: Average eigenvalues of $L(A)$ when training using the airport graph and (1) length-16 walks with memory states (2) length-2 walks with memory states and (3) length-16 walks without memory states. The average AUCs are (1) $.975 \pm .008$ (2), $.968 \pm .005$, (3)$.972 \pm .006$ and average APs are (1)$.977 \pm .007$, (2)$.97 \pm .003$, (3)$.972 \pm .004$.

for LSTMs. Alternative explanations include effectively not having enough parameters: when the memory states are set to zero, a considerable number of the LSTM parameters become useless, and the capacity of the memoryless LSTM may be insufficient.

8.9.4 Network size

We consider varying sizes for the (1) length of the vectors after projecting the size n^* node representations (projection size) (2) number of generator units and (3) number of discriminator units. Following Bojchevski et al. (2018), our baseline projection size is $n' = 64$. The number of hidden units in the generator/discriminator are 40/30 respectively. For our subsequent experiments, we kept the same ratio of hidden units between the generator and the discriminator.

We find that for the Football graph, 40/30 hidden units for the generator/discriminator resulted in higher ROC AUC/AP than either 60/45 or 20/15. This suggests that while 20/15 is not powerful enough to learn as well as 40/30, 60/45 is overfitting. For the Email graph, we did not see evidence of overfitting until we increased the projection size to $n' = 128$, where we see that a

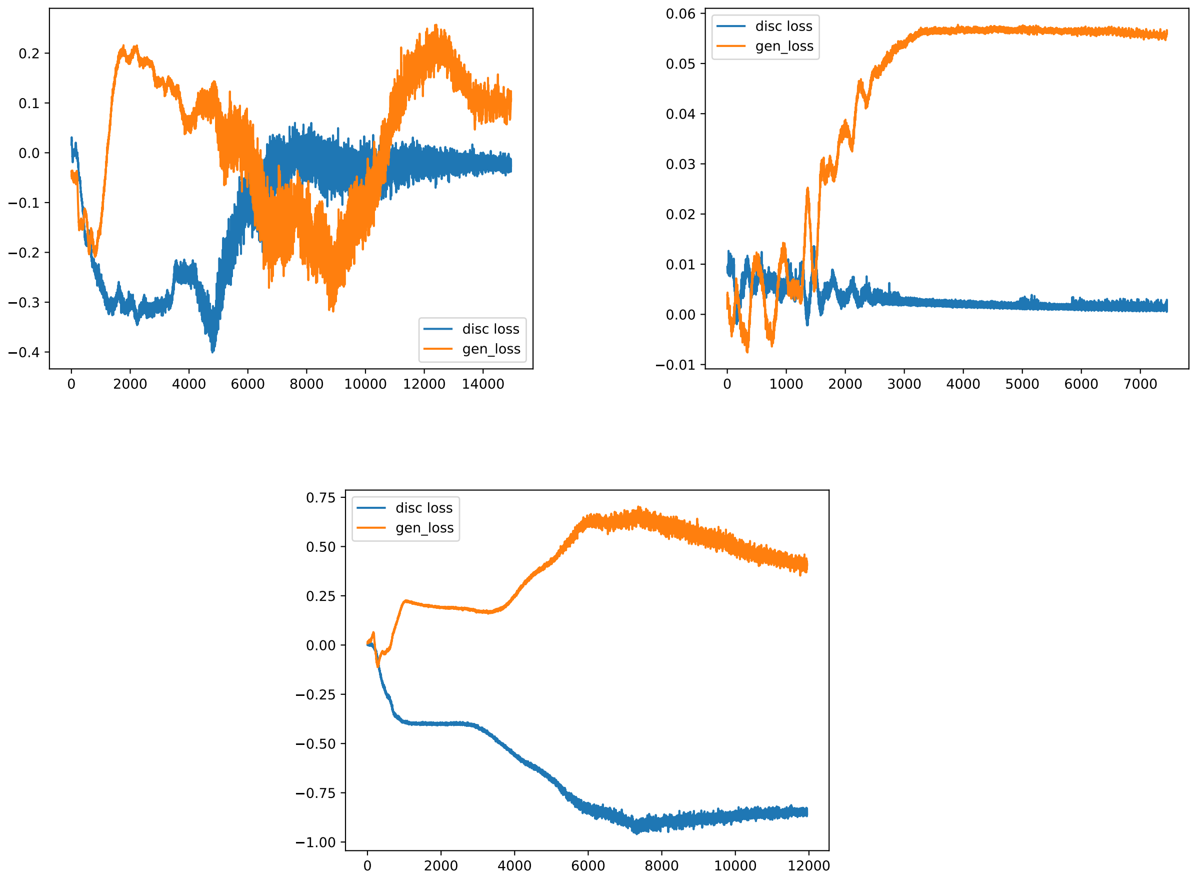


Figure 8.7: Discriminator and generator losses during training using the Five Cluster graph and (1) length-16 walks with memory states, (2) length-2 walks with memory states, and (3) length-16 walks without memory states.

120/90 configuration has lower ROC AUC/AP for $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(T)}) \leq .6$ (Figures 8.8 and 8.9). For the Airport graph, while the 20/15 and 40/30 configurations with projection sizes 64 and 128 are not as powerful as the larger configurations, we do not observe evidence of overfitting. Overall, in our experiments we observe that adding units risks overfitting when training until $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(T)})$ is small.

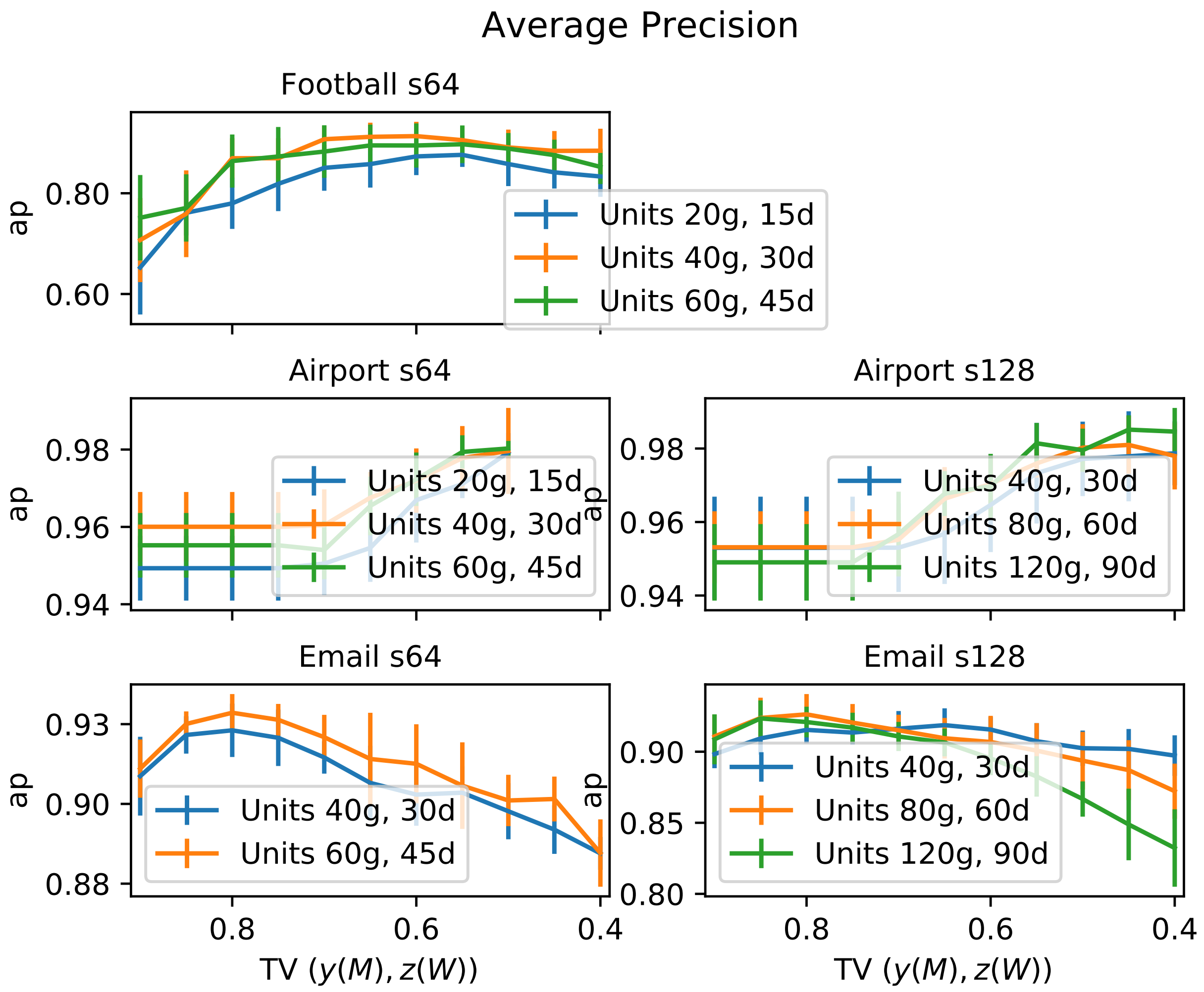


Figure 8.8: Average Precision of frequency matrices learned with different NetGAN sizes.

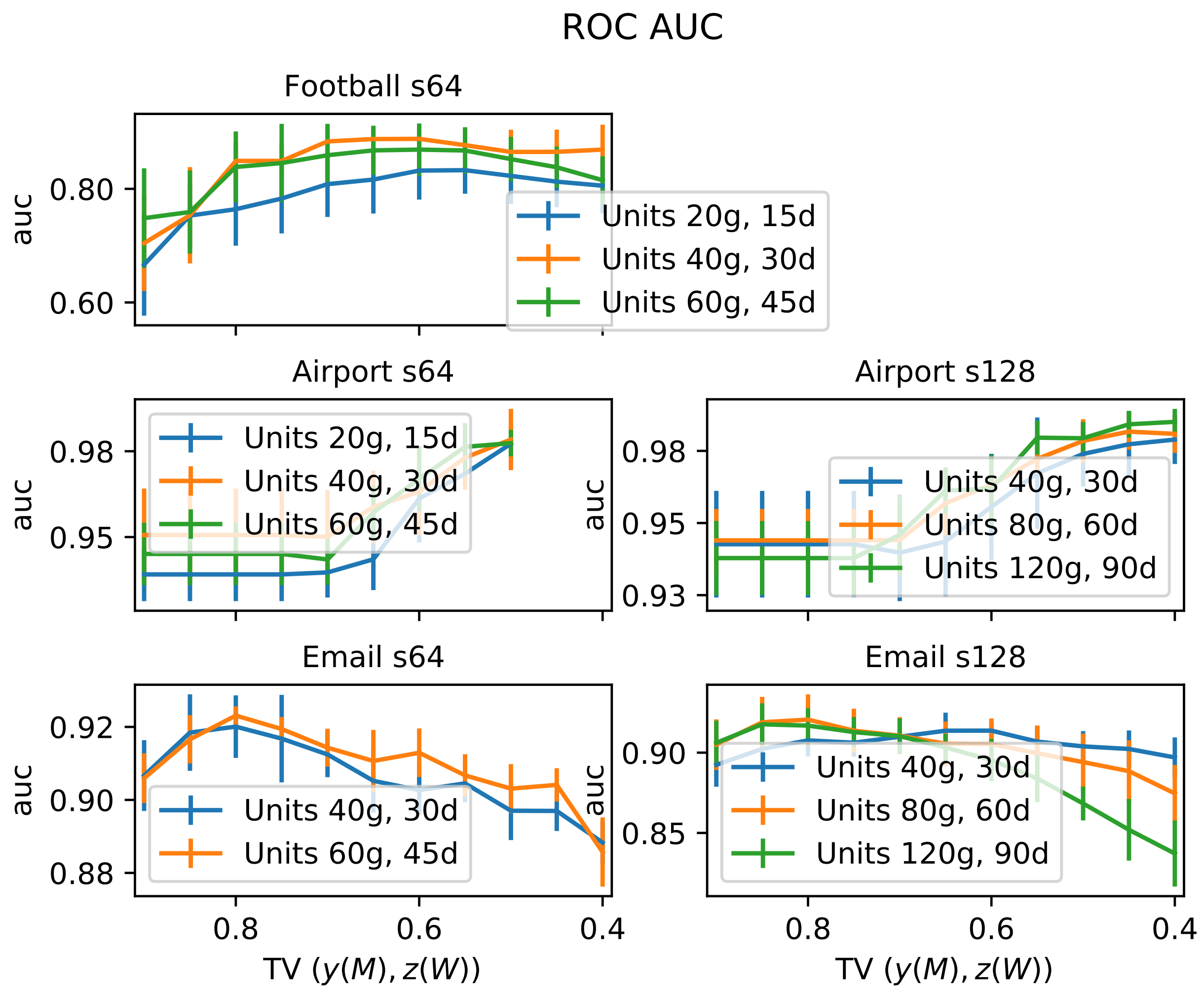


Figure 8.9: ROC AUC for frequency matrices learned with different NetGAN sizes.

Chapter 9

Random Walk Graph Generation

9.1 Motivation

One of the shortcomings of NetGAN graph generation is how opaque it is (Chapter 8). NetGAN trains a GAN to generate synthetic walks and uses the frequencies that edges appear on those walks to build a probabilistic graph which is used as a graph generator using randomized rounding. GAN training is curtailed so that the walk distribution learned is *noisy*; a noisy walk distribution travels between pairs of vertices that are not in $\mathcal{E}^*$. Therefore, by using synthetic walks instead of real walks, edges not in the training set can appear with positive probability in the output graphs. We refer to the distribution adding probability mass on pairs of vertices that do not appear in $\mathcal{E}^*$ as applying *addition noise* (as opposed to removal noise, which is removing probability mass from pairs that do appear in $\mathcal{E}^*$). Experiments show that NetGAN is successful in applying noise to a walk distribution to yield frequency matrices that encode structural properties of the target graph. However, it is difficult to understand *how* by nature of the underlying neural net. Instead, we explore if it is possible to use random walks from the target graph to build graph generators that are easier to understand.

One of the key findings of our NetGAN noise inquiry was identifying where the NetGAN distribution applies addition noise. Compared to applying addition noise uniformly among all pairs not in $\mathcal{E}^*$, the NetGAN distribution applies addition noise less often across the Fiedler cut and more liberally on either side (Section 8.8). Is adding noise on either side of sparse cuts within dense parts, as a means of adding variation to the generated graphs, while preserving cut sparsity the main utility of NetGAN? If so, can we identify sparse cuts and keep them sparse in the resulting graphs from sampled random walks alone, instead of generating graphs from synthetic NetGAN walks trained from these random walks?

9.2 Overview

Random walk generation constructs a probababilistic adjacency matrix A by sampling collections of walks $\mathcal{W}$[1]. All walks in random walk generation are standard random walks (Section 2.4). The guiding principle of generating A is that if two nodes u and v are on opposite sides of a low-conductance cut $S \subseteq \mathcal{V}^*$ (meaning that $u \in \mathcal{S}$, $v \in \overline{\mathcal{S}}$), then two random walks starting at u and v will with high probability take many steps before they meet. Suppose we generate many random walks starting from node u and for all $v \neq u$ we assign $a_{u,v}$ proportional to both (1) the number of times node pairs u and v appeared on the same walk and (2) the number of steps between them on those walks. A walk leaves a cut $\mathcal{S}$ if it starts at a vertex $\boldsymbol{w}[1] = u \in \mathcal{S}$ and $\boldsymbol{w}[t] = v \in \overline{\mathcal{S}}$ for some time t. By the property that random walks take many steps before leaving low-conductance cuts, nodes v that appear in highly-clustered parts of $\mathcal{G}^*$ with u should have high $a_{v,u}$ and those v separated from u by sparse cuts should have small $a_{v,u}$. So if we include edges in $\mathcal{G}$ using randomized rounding on A, clusters of nodes with high edge density in $\mathcal{G}^*$ should with high probability have high edge density in $\mathcal{G}$ (Definition 2.3.1). Our goal is not to memorize the nodes that are connected within these clusters, but to identify them as edge-dense regions and keep these areas dense in the resulting graph. At the same time, if walks take many steps before leaving dense regions then the cuts that are sparse in $\mathcal{G}^*$ should with high probability be sparse in $\mathcal{G}$.

Using this intuition, we construct frequency matrix $\tilde{A}$ in rounds that each have two phases.

1. In the first phase, we generate b walk lists $\mathcal{WS} = \mathcal{W}_1, \mathcal{W}_2 \ldots \mathcal{W}_b$. Each list of walks $\mathcal{W}_i$ is built with respect to a random cut $\mathcal{S}$ that is defined from a random walk generation process. We call this the *cut construction* phase (Section 9.4).

2. In the next phase, for each $\mathcal{W}_i \in \mathcal{WS}$ we update $\tilde{A}$ for each $\boldsymbol{w} \in \mathcal{W}_i$. Entries of $\tilde{A}$ are increased based on how many times pairs (u, v) appear on walks $\boldsymbol{w} \in \mathcal{W}_i$, how many steps u and v are apart on the walks, and whether or not they are both in $\mathcal{S}$ or $\overline{\mathcal{S}}$ (Section 9.5).

[1]Probabilistic adjacency matrix defined in Section 2.2)

After a sufficient number of rounds are completed, $\tilde{A}$ is scaled to construct A so that $\sum a_{u,v} = m^*$ and $a_{u,v} \in [0, 1]$ (Algorithm 14).

Algorithm 16: Random walk generation.

```
Result: Probabilistic graph $A$
Input: Input graph $\mathcal{G}^*$;
Number of walk iterations per batch $b$;
Minimum number of nodes needed to be seen by initialization walks $k$;
Whether or not to add mass across cuts zeroCross;
Whether or not to weight the amount of mass added by the number of steps between nodes
walkDistWeight;
Total number of walk algorithm iterations $T$;
Initialize $\tilde{A} = 0_{n^*}$ ;
for round $\leftarrow$ 1 to $T/b$ do
    Construct cut $\mathcal{S}$ and list of walks $\mathcal{WS}$ from cutConstruct($\mathcal{G}^*, b, k$) (Algorithm 18) ;
    $\tilde{A} = \text{matrixUpdate}(\tilde{A}, \mathcal{S}, \mathcal{WS}, \text{zeroCross}, \text{walkDistWeight})$ (Algorithm 19);
end
Construct probabilistic graph $A = \text{scaleMat}(\tilde{A}, m^*)$ (Algorithm 14) ;
```

9.3 Related work

Our problem of assigning probability $a_{u,v}$ to include (u, v) in $\mathcal{E}$ is similar to the *link prediction* problem (Liben-Nowell and Kleinberg, 2007). The link prediction problem is to predict whether nodes in a graph should have an edge between them. Not only is assigning a probability that a link should exist similar to the problem of assigning edge probabilities, but a long line of work uses random walks for link-prediction (Backstrom and Leskovec, 2011; Grover and Leskovec, 2016; Yin et al., 2010). One of the algorithms is *Node2Vec* which uses random walks to learn node embeddings from maximizing the likelihood that these embeddings preserve neighbors sampled from random walks (Grover and Leskovec, 2016).

Random walk methods to partition a graph is not a new concept. Nibble by Spielman and Teng (2013) uses random walks to partition a graph into clusters such that each cluster has low conductance. The number of steps it takes a random walk to leave a cut is related to the conductance of a cut and is one of the key properties behind our approach of using random walks to discover sparse cuts. Spielman and Teng (2013) use the "lazy" random walk that uses the standard random walk with probability $\frac{1}{2}$ and stays at the current node with probability $\frac{1}{2}$. For a lazy walk of length t that starts in a cut $\mathcal{S}$, the probability that the walk stays entirely in $\mathcal{S}$ for all steps is lower bounded by $1 - t\varphi_{\mathcal{G}^*}(\mathcal{S})/2$ (Proposition 5.0.1). Following Spielman and Teng (2013), Andersen et al. (2006) improve Nibble by using the *page-rank vector* that circumvents the need to generate many random walks to partition the graph because the page-rank vector encodes information about the *hitting-times* of random walks. The hitting-time between two vertices in a random walk is the expected number of steps before a random walk starting at one random node will hit the other. The hitting-time is proportional to the sparsity of cuts separating the two nodes.

Another generative graph model that builds off the idea of applying variation in random graphs within regions that are highly clustered in the target is *MUSKETEER* (Gutfraind et al., 2015). MUSKEETEER generates a random graph by coarsening the target $\mathcal{G}^*$ to find dense regions and then makes randomized edge edits during the uncoarsening stage to generate a random graph.

9.4 Cut Construction stage

Each round constructs cut $\mathcal{S} \subseteq \mathcal{V}^*$ with a batch of walk lists $\mathcal{WS}$. The i-th walk list in $\mathcal{WS}$ is denoted as $\mathcal{W}_i$ and all $\mathcal{W}_i$ are constructed using the same two independent random walks $\boldsymbol{q}[1]$ and $\boldsymbol{q}[2]$ on $\mathcal{G}^*$ which we call *seed walks*. We run the seed walks from two different vertices until one hits a vertex the other has in previous steps or they intersect at some step t. At this point, we label the first $t-1$ nodes v hit by $\boldsymbol{q}[1]$ with $f(v) = 1$ and the first $t-1$ nodes v hit by $\boldsymbol{q}[2]$ as $f(v) = 2$. The sampling of the seed walks is in Algorithm 17.

After the seed walks are terminated, we continue adding walks to $\mathcal{W}_i$ until all nodes are seen at least once or we hit a maximum number of walks. While sampling random walks, we maintain two voting maps g_1 and g_2 that map $\mathcal{V}^*$ to a number of votes. Map g_1 initially maps each node in $\boldsymbol{q}[1]$ to 1 and all others to 0. Similarly, g_2 initially maps each node in $\boldsymbol{q}[2]$ to 1 and all others to 0. We sample random walks to add to $\mathcal{W}_i$ by initializing walks from unmarked nodes. Initially, only the nodes on the nodes labeled by the seed walks are marked. We run those random walk until we hit a node labeled by a seed walk. If the walk hits a node u with $f(u) = 1$, then we increment

```
Algorithm 17: sampleSeedWalks
  Result: $\boldsymbol{q}[1]$, $\boldsymbol{q}[2]$ ;
  Input: $\mathcal{G}^*$;
  Minimum number of nodes needed to be seen by initialization walks $k$;
  Set ranSeedWalksFlag $= 0$ ;
  $f(v) = 0$ for all $v \in \mathcal{V}^*$ ;
  while (Number of nodes $v$ such that $f(v) = 1$ is less than $k$) or (Number of nodes $v$ such
   that $f(v) = 2$ is less than $k$) do
      if ranSeedWalksFlag is 1 then
          reset $f(v) = 0$ for all $v \in \mathcal{V}^*$ ;
      Choose random nodes $u \neq v$ with probability proportional to $d_u, d_v$ ;
      $t = 1$ ;
      Initialize seed walks $\boldsymbol{q}[1](t) = u$, $\boldsymbol{q}[2](t) = v$ ;
      while $\boldsymbol{q}[1](t)$ is not equal to $\boldsymbol{q}[2](t)$ do
          if $f(\boldsymbol{q}[1](t)) = 2$ then
              break.
          if $f(\boldsymbol{q}[2](t)) = 1$ then
              break.
          $f(\boldsymbol{q}[1](t)) = 1$ ;
          $f(\boldsymbol{q}[2](t)) = 2$ ;
          $t = t + 1$ ;
      Set flag ranSeedWalksFlag $= 1$ ;
```

$g_1(v)$ for all nodes v traversed on the walk (as many times as it appears on the walk). Else the walk hits a node u labeled with $f(u) = 2$ first and we increment $g_2(v)$ for all nodes v traversed by the walk. Once the walk hits a labeled node, we terminate that walk and add it to $\mathcal{W}_i$. All the nodes on the walk are now marked. This is repeated until all nodes are marked or we reach a maximum number of walks ℓ, this completes the construction of walk list $\mathcal{W}_i$. We build walk lists in this manner until all b walk lists are finished at which point the nodes are divided according g_1 and g_2. If $g_1(v) > g_2(v)$, then v was on more walks that hit nodes labeled with 1 than those that hit nodes labeled with 2. All nodes with $g_1(v) > g_2(v)$, including those labeled with 1 by $\boldsymbol{q}[1]$ are placed in $\mathcal{S}$. The full cut-construction algorithm is in Algorithm 18.

Our goal is that by stopping walks once they hit one of the labeled nodes that this discovers sparse cuts. Suppose there exists a $\mathcal{S} \subseteq \mathcal{V}^*$ such that all nodes v with $f(v) = 1$ are in $\mathcal{S}$ and all nodes $f(v) = 2$ are not. Then due to the sparsity of $\mathcal{S}$, then we would expect most walks starting at the unseen nodes $u \in \mathcal{S}$ will hit a labeled node in $\mathcal{S}$ before one that is not. The idea is if there are enough walks in $\mathcal{WS}$ then we should be able to find $\mathcal{S}$ if there is such a sparse cut separating the

Algorithm 18: cutConstruction

```
Result: $\mathcal{S} \subseteq \mathcal{V}^*$, $\mathcal{WS}$
Input: $\mathcal{G}^*$;
Number of walk iterations per batch $b$;
Minimum number of nodes needed to be seen by initialization walks $k$;
Maximum number of walks per iterations $\ell$;
Sample initial disjoint walks:
Sample seed walks $\boldsymbol{q}[1]$ and $\boldsymbol{q}[2]$ that are stopped once one hits a node the other has seen
 (Algorithm 17) ;
Initialize votes:
foreach $v \in \mathcal{V}^*$ do
    if $f(v) = 1$ then
        $g_1(v) = 1$ and $g_2(v) = 0$ ;
    else if $f(v) = 2$ then
        $g_2(v) = 1$ and $g_1(v) = 0$ ;
    else
        $g_1(v) = g_2(v) = 0$ ;
    end
end
Build list of walk lists:
$\mathcal{WS} = \emptyset$, $V' = \{v \in \mathcal{V}^* s.t. f(v) = 0\} \neq \emptyset$ ;
if $V' = \emptyset$ then
    Let $\mathcal{WS} = [[\boldsymbol{q}[1], \boldsymbol{q}[2]]]$. Return $\mathcal{WS}$ and $\mathcal{S} = \{v \in \mathcal{V}^* s.t. f(v) = 1\}$ ;
for $j \leftarrow 1$ to $b$ do
    Initialize $\mathcal{W}_j = [\boldsymbol{q}[1], \boldsymbol{q}[2]]$ ;
    while $V' = \{v \in \mathcal{V}^*$ s.t. $f(v) = 0\} \neq \emptyset$ and $|\mathcal{W}_j| < \ell$ do
        Choose random node from $V'$ that takes value $v$ probability $d_v/Z$ with
         $Z = \sum_{v \in V'} d_v$ ;
        $t = 1, \boldsymbol{w}(t) = v$ ;
        Walk until $\boldsymbol{w}$ hits a labeled node
        while $f(\boldsymbol{w}(t)) = 0$ do
            $t = t + 1$ ;
        end
        for $i \leftarrow 1$ to $t - 1$ do
            if $f(\boldsymbol{w}(t)) = 1$ then
                $g_1(\boldsymbol{w}(i)) = g_1(\boldsymbol{w}(i)) + 1$ ;
            else
                $g_2(\boldsymbol{w}(i)) = g_2(\boldsymbol{w}(i)) + 1$ ;
        end
        Append $\boldsymbol{w}$ to $\mathcal{W}_j$ ;
    end
    Append $\mathcal{W}_j$ to $\mathcal{WS}$;
end
Construct cut using votes:
$\mathcal{S} = \{v \in \mathcal{V}^*$ s.t. $g_1(v) > g_2(v)\}$ ;
Return $\mathcal{S}$ and $\mathcal{WS}$;
```

labeled nodes.

One of the key questions in implementing this algorithm is how large does $\mathcal{WS}$ need to be to discover enough cuts and see enough of the graph? Does it matter how large the b is to find a meaningful cuts? To help compare the effects of different b, we normalize the number of walk lists $\mathcal{W}$ we sample by running T/b number of rounds. In each round, we will sample b number of $\mathcal{W}$ unless the initialization random walks label every node. Therefore, each round defines a $\mathcal{S}$ using enough walks such that we have either (1) Discovered a cut sparse enough such that the two walks hit every node on its side before crossing or (2) Sampled ℓ walks or seen every node at least once b number of times.

9.5 Constructing frequency matrix from walk and cut collections

Algorithm 19: matrixUpdate

```
Result: Frequency matrix $\tilde{A}$ updated for pairs of nodes in walks in $\mathcal{WS}$
Input: Frequency matrix $\tilde{A}$;
Cut $\mathcal{S}$;
List of walk lists $\mathcal{WS}$;
Whether or not to add mass across cuts zeroCross;
Whether or not to weight the amount of mass added by the number of steps between nodes
walkDistWeight;
for $\mathcal{W} \in \mathcal{WS}$ do
    for $\boldsymbol{w} \in \mathcal{W}$ do
        If $\boldsymbol{w}(1) \in \mathcal{S}$, let $\ell'$ be the smallest integer such that $\boldsymbol{w}(\ell'+1) \notin \mathcal{S}$ if $\boldsymbol{w}(\ell'+1) \notin \mathcal{S}$
          exists and $length(\boldsymbol{w})$ otherwise. Else, let $\ell'$ be the smallest integer such that
          $\boldsymbol{w}(\ell'+1) \in \mathcal{S}$ if $\boldsymbol{w}(\ell'+1) \in \mathcal{S}$ exists and $length(\boldsymbol{w})$ otherwise. ;
        If zeroCross, $\ell = \min(\ell', \text{maxDist})$. Else, $\ell = \min(length(walk), \text{maxDist})$ ;
        for $z \leftarrow 1$ to $\ell - 1$ do
            If walkDistWeight, $q = 1/z$. Else, $q = 1$. For $t = 1, 2, \dots \ell - z$, let $u = \boldsymbol{w}(t)$ and
              $v = \boldsymbol{w}(t+z)$ ;
            $\tilde{a}_{u,v} = \tilde{a}_{u,v} + q$, $\tilde{a}_{v,u} = \tilde{a}_{v,u} + q$ ;
        end
    end
    For all $v \in \mathcal{V}^*$, $\tilde{a}_{v,v} = 0$ ;
end
```

After constructing $\mathcal{S}$ and $\mathcal{WS}$, we enter the *matrix update* stage of the round and add value to entries of $\tilde{A}$. For each walk $w_j \in \mathcal{W}_i \in \mathcal{WS}$, we update $\tilde{A}$ by adding positive q to $\tilde{a}_{u,v}$ for pairs $(u, v) = (w_j(t), w_j(t+z))$ where $w_j(t)$ is the node hit at time t by w_i. The amount q added to $\tilde{a}_{u,v}$

can be a function of z so that nodes that appear closer in w_i are added more weight (*walk distance weight*). Another variant we explore is non-zero q only for $t \leq \ell'$ where ℓ' is the step that w_i crosses from $\mathcal{S}$ to $\overline{\mathcal{S}}$ or vice versa (*zero-cross*). This ensures that if our walks have succeeded in finding a sparse cut, our matrix update does not add any mass across it. The variants we try for matrix updating our summarized in the list below:

- Default: Assign $q(w_j, t, z) = 1$ for all w_j, t, z.
- Zero Cross (zc): Without loss of generality, suppose $w_j(0) \in \mathcal{S}$. If all the nodes in w_j are in $\mathcal{S}$, $q(w_j, t, z) = 1$ for all t, z. Else, let ℓ' be the smallest ℓ' such that $w_j(\ell') \in \overline{\mathcal{S}}$. Then $q(w_i, t, z) = 1$ for all $t + z < \ell'$ and 0 otherwise.
- Walk Distance Weight (wdw): Assign $q(w_j, t, z) = 1/z$ for all w_j, t.
- Zero Cross and Walk Distance Weight (zc+wdw): Assign $q(w_j, t, z) = 1/z$ for all $t + z < \ell'$ and 0 otherwise with ℓ' defined as above.

9.6 Related graph measures

In order to better interpret what random walk generation is capturing, we turn to non-random matrix models computed from properties that appear related to the mechanisms used.

The walk distance weights in the matrix update phase are akin to a weight computed from by shortest path distance. The shortest path distance $\mathrm{sp}(v, u, \mathcal{G}^*)$ between nodes v and u on $\mathcal{G}^*$ is the length of the shortest path connecting v and u. When constructing $\tilde{A}$, if pairs v and u appear on many of the same walks then their distance along the walk is likely to be proportional to their shortest path distance. Therefore, since our walk distance weight assigns weight inversely proportional to the number of steps between pairs on walks we are interested to see if we could compute the weights from shortest paths alone instead of using random walks. We compute shortest path distance matrices $\widetilde{SPM}$ with inverse powers k so that $\widetilde{spm}_{u,v} = (1/sp(u, v, \mathcal{G}^*))^k$.

By using random walks that are terminated early, it seems natural to assume that pairs (u, v) that have small *average commute time* would appear more often and closer together on the same walks then those that don't. Let $c'(u \mid v, \mathcal{G}^*)$ be the expected number of steps a walk starting at v takes before hitting u for the first time. The average commute time of a pair (u, v) in $\mathcal{G}^*$

is $ct(u, v, \mathcal{G}^*) = c'(u \mid v, \mathcal{G}^*) + c'(v \mid u, \mathcal{G}^*)$ so that $ct(v, u, \mathcal{G}^*) = ct(u, v, \mathcal{G}^*)$. As we expect the frequencies v and u appear on the same walk to be inversely proportional to commute time, we compute $\widetilde{CTM}$ with inverse power k so that $\widetilde{ctm}_{u,v} = (1/ct(u, v, \mathcal{G}^*))^k$. To compute the average commute time in $\mathcal{G}^*$, we use the pseudo-inverse of the the non-normalized laplacian of A^* Fouss et al. (2006).

For both shortest path and average commute time, we compute probabilistic graphs SPM and CTM computed from $\widetilde{SPM}$ and $\widetilde{CTM}$ to A generated from random walk generation. We compute SPM and CTM by scaling $\widetilde{SPM}$ and $\widetilde{CTM}$ using Algorithm 14 to construct a probabilistic graph so that we can compare the utility of these probabilistic graphs to those generated using Random walk generation.

9.7 Experiments

We experiment with different batch sizes to see how large the batch size should be to help produce sparse cuts (Table 9.1). We observe that increasing the batch size seems to slightly help to discover cuts with smaller conductance.

We find that using the zero cross rule and the walk distance weight rule is most effective in matching laplacian spectra in expectation (Figure 9.2). In particular, the walk distance weight rule is effective in helping match laplacian spectra. This however comes at an entropy cost (Figure 9.3).

We plot the eigenvalues of A across our various hyper-parameters and shortest path models and commute time models (SPM and CTM) for different inverse powers k in Figure 9.4. As k grows, SPM and CTM capture more spectral structure (Figure 9.7). This is likely because as k grows, the weights for pairs (v, u) longer shortest paths and commute times goes to zero which helps preserve weights on only pairs in $\mathcal{E}^*$.

We plot the smallest integer k for which the spectral performance measured by $\ell_2^{\mathrm{LW}}(\lambda^*, \lambda(SPM))$ is at least as small as $\ell_2^{\mathrm{LW}}(\lambda^*, \lambda(A))$ for A generated with walk-distance weight and zero cross updates with a batch size of 10. For the graphs where SPM and CTM had similar spectral performance to A, the entropy was also comparable (Figures 9.5, 9.6). However, for some graphs the spectral performance of CTM could not match that of A regardless of k (e.g. Airport CTM and Email CTM, Figure 9.7). For the Netscience graph, the smallest k for CTM and SPM that matched

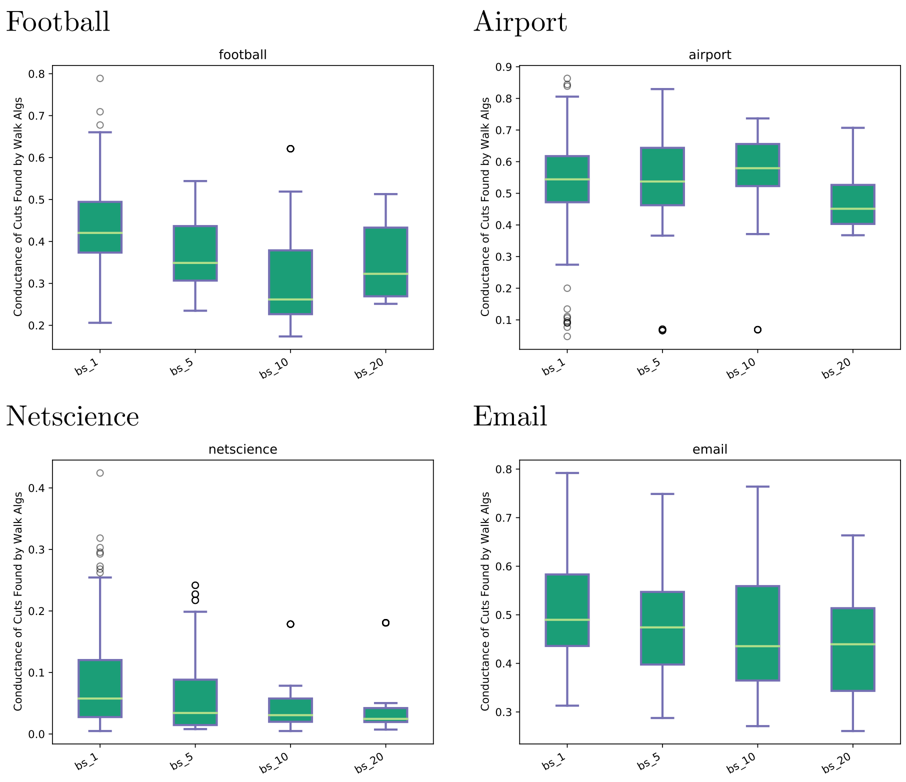
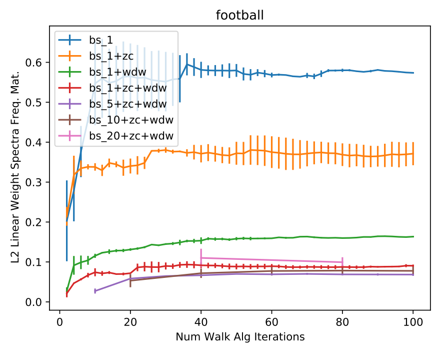
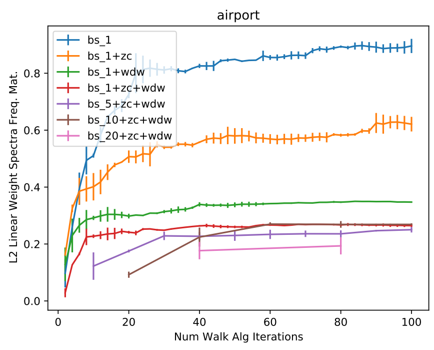


Figure 9.1: Conductance of cuts found via Random walk generation. We observe that a larger batch size finds cuts with smaller conductance on average.

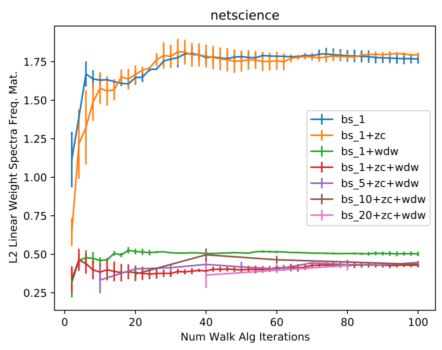
Football



Airport



Netscience



Email

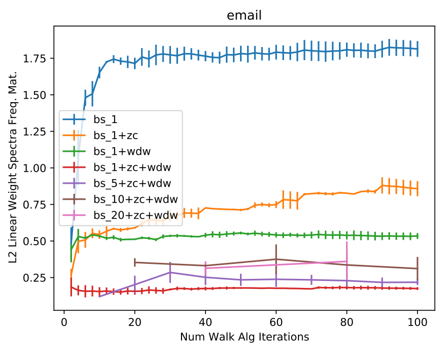
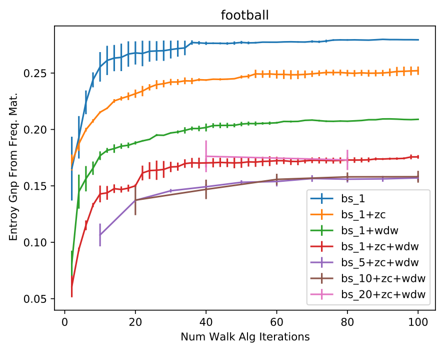
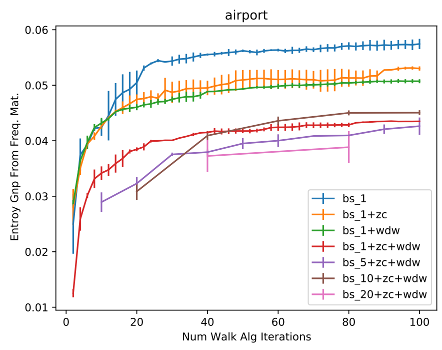


Figure 9.2: Difference between true spectrum and spectrum of probababilistic adjacency matrix A measured by ℓ_2^{LW} after $T = 100$ sets of walks for different Random walk generation parameters.

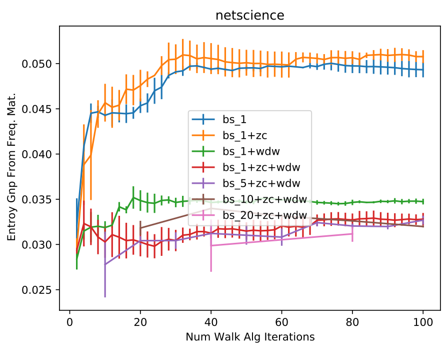
Football



Airport



Netscience



Email

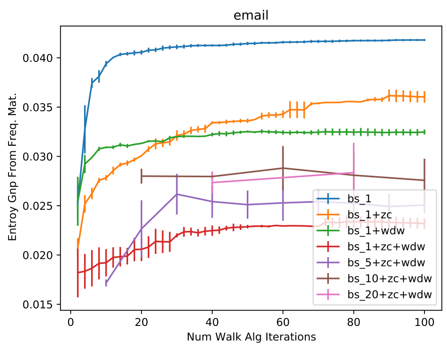


Figure 9.3: Mean entropy of probababilistic adjacency matrix entries $a_{u,v}$ after $T = 100$ sets of walks for different Random walk generation parameters.

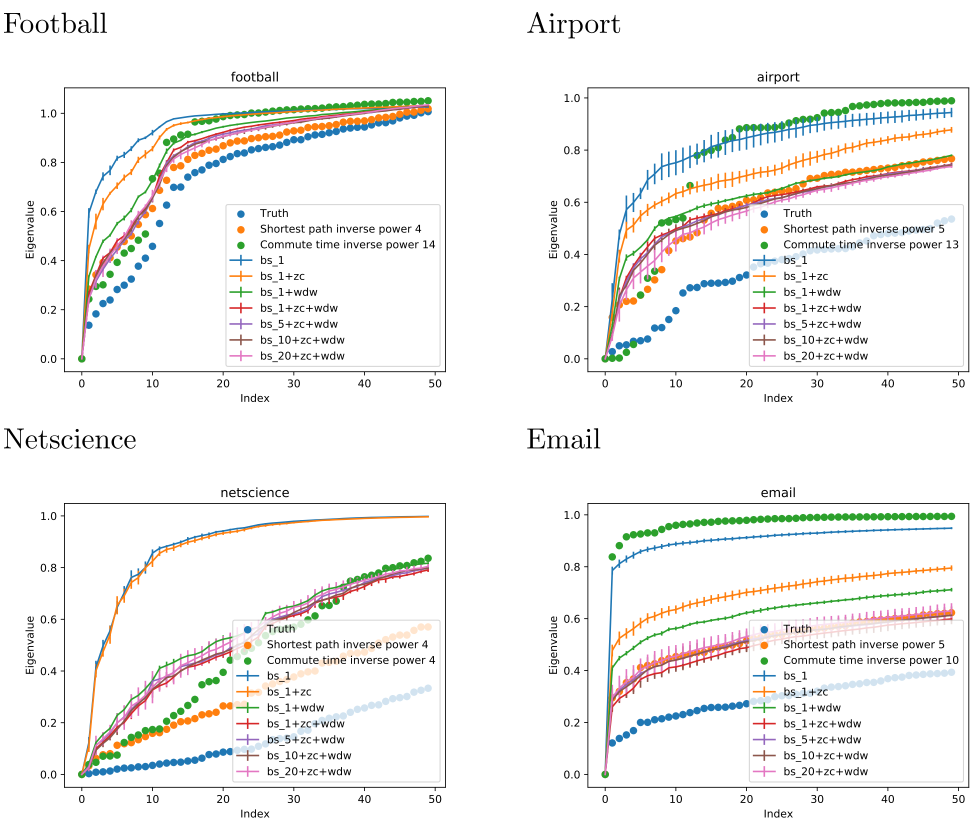


Figure 9.4: Eigenvalues of probababilistic adjacency matrices after 100 walk algorithm iterations compared to inverse of shortest path matrix.

the spectral performance of A beats the spectral performance of A by a significant margin which is accompanied by a drop in entropy. The main take away from using shortest path distances and average commute times is that while for appropriate k the SPM seems to be comparable to random walk generation, using commute times to build frequency matrices CTM does not match spectra as well in general.

Graph	k	Entropy
Football	4	.145
Netscience	4	.003
Airport	5	.054
Email	5	.023

Figure 9.5: Entropy of shortest path model SPM with entries $spm_{u,v} = (1/SP(\mathcal{G}^*, u, v))^k$.

Graph	k	Entropy
Football	14	.149
Netscience	4	.003
Airport	13	.006
Email	10	.01

Figure 9.6: Entropy of commute time model CTM with entries $ctm_{u,v} = (1/CT(\mathcal{G}^*, u, v))^k$.

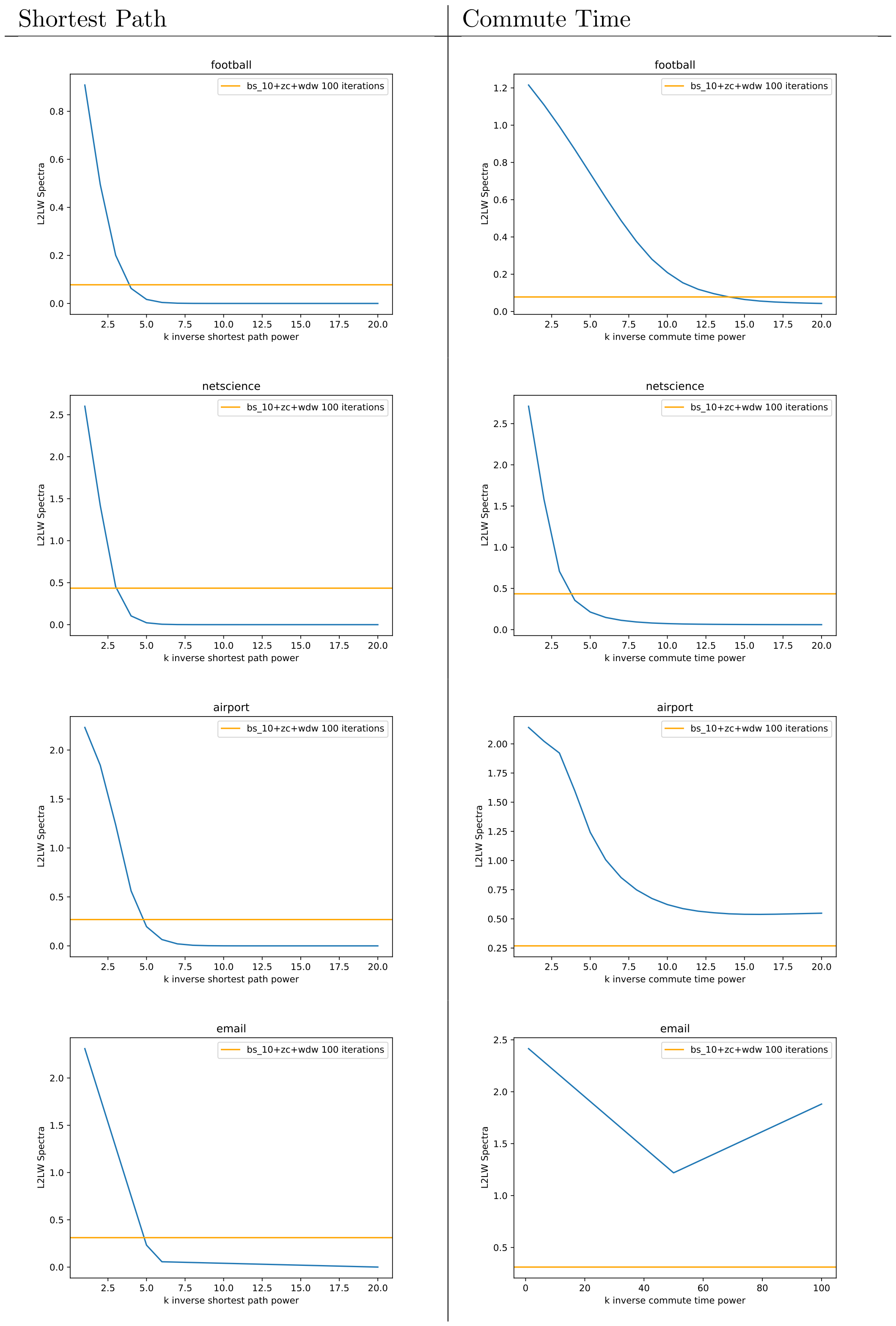


Figure 9.7: Difference between true spectrum and spectrum of shortest path and commute time probababilistic adjacency matrices meaured by ℓ_2^{LW} for growing inverse power k.

Chapter 10

Cut Fix Graph Generation

Connectivity across cuts is one of the global graph features explored throughout this work that capture graph structure (cut connectivity is defined in Definition 5.0.1). In fact, matching the connectivity across all cuts in A^* characterizes $\mathcal{G}^*$ exactly. Cut fix generation chooses a subset of cuts to match according to some *importance* function in order to capture enough structure of $\mathcal{G}^*$ while still providing diversity. Spectral generation (Chapter 7), the NetGAN with the fastest mixing Markov chain (Chapter 8), and Random walk generation (Chapter 9) all place an emphasis on sparse cuts in A^* because dense cuts are easy to achieve with the most basic Erdős Rényi type random graph models. Alternatively, Cut fix generation chooses cuts to focus on by starting with a naive baseline probabababilistic adjacency matrix A_0 and identifies cuts that approximately differ the most from A^* in terms of the number of edges crossing the cut. Once a cut $\mathcal{S}$ has been identified, it is "corrected" by making edge additions/deletions so that the connectivity across each $\mathcal{S}$ matches that in A^*. The idea is that once enough cuts are corrected, the final $\mathcal{G}$ generated from randomized rounding (Definition 2.3.1) should have structure similar to $\mathcal{G}^*$.

Recall from Definition 5.0.1 that the connectivity of $\mathcal{S}$ with respect to A measures the sum of probabilistic edges crossing the cut: $\partial(\mathcal{S}, A) = \sum_{v \in \mathcal{S}, u \in \overline{\mathcal{S}}} a_{v,u}$. A *cut correction* on $\mathcal{S}$ changes the entries in A so that $\partial(\mathcal{S}, A) = \partial(\mathcal{S}, A^*)$. The goal is to adaptively choose cuts to correct in A with entries $a_{v,u} \in [0, 1]$ to match A^* in order to improve the performance of A as a model for $\mathcal{G}^*$ with respect to $\ell_2^{\mathrm{LW}}(\lambda(A^*), \lambda(A))$ or some other metric. To see why this might be helpful, consider the barbell-graph as $\mathcal{G}^*$ with one edge connecting a cut $\mathcal{S}^*$ and $\overline{\mathcal{S}}^*$. For any A, if the connectivity of $\mathcal{S}^*$ is corrected to have a single edge crossing and the remaining edge mass on $\mathcal{S}^* \times \mathcal{S}^*$ and $\overline{\mathcal{S}}^* \times \overline{\mathcal{S}}^*$ then A becomes a reasonable model of $\mathcal{G}^*$. Once A is corrected, $\mathcal{G}$ is generated by including each pair (v, u) in $\mathcal{E}$ with probability $a_{v,u}$.

A cut correction algorithm requires the following:

1. A method of choosing the cuts to correct. Which cuts are the most important to capture the important properties of $\mathcal{G}^*$?

2. Construction of the perturbation to add to A that corrects $\mathcal{S}$ to match A^*. The construction controls the entropy change to A: if A has high-entropy values near .5, then a perturbation that pushes them towards 0 and 1 will lower the entropy of the distribution that we draw $\mathcal{G}$ from.

3. A number of cuts to correct. If we correct too many, we could end up with little variability among $\mathcal{G}$.

The Cut fix generation algorithm is in Algorithm 20.

Algorithm 20: Cut fix generation.

```
Result: Probabilistic adjacency matrix $A$ with cuts matched to have connectivity that
        matches $A^*$.
Input: Input adjacency matrix $A^*$;
Probabilistic adjacency matrix $A$;
Number of cuts to correct $k$;
Neighborhood size for $GRASP$ cut sampling $\alpha$;
for $i \leftarrow 1$ to $k$ do
    $\mathcal{S} = approxGrasp(A^*, A, \alpha)$ ;
    $A = cutCorrect(A^*, A, \mathcal{S})$ ;
end
```

10.0.1 Cut selection: selecting cuts that differ the most from target

Cut fix generation corrects cuts with connectivity that differs the most in A from A^*. These cuts are an approximate solution to max $|\partial(\mathcal{S}, A) - \partial(\mathcal{S}, A^*)|$ using a *Greedy Randomized Adaptive Search Procedure* (GRASP) over all $\mathcal{S} \subseteq \mathcal{V}^*$ so that the connectivity across $\mathcal{S}$ is either much higher or lower in A than in A^*. Expanding the objective function,

$$\begin{aligned}|\partial(\mathcal{S},A)-\partial(\mathcal{S},A^*)| &= |(\sum_{v\in\mathcal{S},u\in\overline{\mathcal{S}}} a_{v,u}) - (\sum_{v\in\mathcal{S},u\in\overline{\mathcal{S}}} a^*_{v,u})| \\ &= |\sum_{v\in\mathcal{S},u\in\overline{\mathcal{S}}} a_{v,u}-a^*_{v,u}| \\ &= |\partial(\mathcal{S},A-A^*)| \qquad (10.1)\end{aligned}$$

The goal is to find the $\mathcal{S}$ that maximizes Equation (10.1). The optimization problem can be re-framed as finding $\mathcal{S}$ with high connectivity (positive) or low connectivity (negative) in a weighted graph with adjacency matrix $(A-A^*)$ with negative edge weights.

10.0.2 Cut correction: keeping expected edges constant

Once a $\mathcal{S}$ is chosen to correct, it is corrected in A using a symmetric perturbation matrix E. Matrix E is constructed so that $\partial(\mathcal{S},A+E)=\partial(\mathcal{S},A^*)$ and $a_{v,u}+e_{v,u}\in[0,1]$. Perhaps the simplest method to correct $\mathcal{S}$ would be to uniformly remove/add the amount needed from all entries $(v,u)\in(\mathcal{S}\times\overline{\mathcal{S}})$ (without pushing entries outside $[0,1]$). However, we add the additional goal of constructing A so that approximately m^* number of edges are included in $\mathcal{G}$. We construct E with $\sum_{u,v} e_{v,u}=0$ so that if we start with $\sum_{u,v} a_{v,u}=m^*$ then the m^* expected number of edges does not change once we add E to A.

10.1 Related Work

Using cuts to define similarity between matrices is not a new problem. (Frieze and Kannan, 1999) provide an algorithm to find a sum of rank-1 matrices that approximate a matrix by approximately matching the sum of entries in all sub-matrices (and consequently approximately matching all cuts). Within the context of graphs, Borgs et al. (2008) provide a relationship between the largest cut connectivity difference between two graphs (cut distance) and the number of adjacency preserving vertex maps between a sub-graph and these two graphs, another notion of graph similarity.

Cut fix generation is related to broader generative graph model technique called *editing* which takes a baseline model and then makes corrections, usually in the form of edge additions and deletions, in order to make progress towards building a model that generates realistic graphs. Perhaps the most classical example of an editing generative graph model is edge swaps which keeps degrees constant by randomly sampling two edges (v_1, u_1) and (v_2, u_2) and replacing them with (v_1, u_2) and (v_2, u_1). More recently, *ReCon* by Staudt et al. (2017) randomizes edges within/between communities while keeping degrees constant. *MUSKEETER* by Gutfraind et al. (2015) also randomizes edges within communities by coarsening a graph and randomizing within the coarsened clusters before projecting the graph back up.

10.2 An approximate *GRASP* procedure to select cuts

10.2.1 Greedy randomized adaptive search procedure (*GRASP*)

We use a *Greedy randomized adaptive search procedure (GRASP)* to find an approximate max solution for the objective function $|\partial(\mathcal{S}, A - A^*)|$ over $\mathcal{S} \subseteq \mathcal{V}^*$ (Feo and Resende, 1995). *GRASP* algorithms are used in combinatorial optimization problems with problem structure that can be exploited to limit the search space and prevent local optima. *GRASP* algorithms consist of two main components: greedy construction and a local search (Algorithm 21).

Algorithm 21: Greedy randomized adaptive search procedure (Feo and Resende, 1995)

```
Result: bestSolution
Input: inputInstance ;
while stopping condition not met do
    solution = constructGreedy(inputInstance) ;
    solution = localSearch(solution, inputInstance) ;
    bestSolution = updateBestSolution(solution) ;
end
```

Local search uses a neighbor function which maps a solution to a set of neighbor feasible solutions (Algorithm 22). The current solution is updated to one of its neighbors that are better according to an objective. Local search stops once we have found a locally optimal solution — a solution for which all of its neighbors are the same or worse than the current solution.

Algorithm 22: *GRASP* local search. Feo and Resende (1995)

```
Result: solution
Input: inputInstance, solution, Neighbor function $N$ ;
while solution not locally optimal do
    Find better solution in $N$(solution) ;
    Update solution ;
end
```

10.2.2 *GRASP* for max/min cut

We use a *GRASP* algorithm to find an approximate max solution for the objective function $|\partial(\mathcal{S}, A - A^*)|$ over $\mathcal{S} \subseteq \mathcal{V}^*$. In both the greedy construction and local search phase, $\mathcal{S}$ is modified by adding/removing nodes starting with the empty set. For the rest of this chapter, we write $W = A - A^*$ for the difference of A and A^*.

The greedy construction phase begins with an initialization stage. We permute $\mathcal{V}^*$ and the first $\beta \cdot n^*$ nodes in permutation order are randomly placed into two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ each with probability $\frac{1}{2}$ where $\beta \in [0, 1]$. Sets $\mathcal{S}_1$ and $\mathcal{S}_2$ initialize $\mathcal{S}$ and $\overline{\mathcal{S}}$. We place the remaining nodes $v \in \mathcal{V}^* \setminus (\mathcal{S}_1 \cup \mathcal{S}_2)$ greedily by maximizing the absolute value of the *vertex connectivity* of v with respect to W:

Definition 10.2.1. Vertex connectivity of v to $\mathcal{S}$ in W Let W be a $n^* \times n^*$ real symmetric matrix, $\mathcal{S} \subseteq \mathcal{V}^*$, and $v \in \mathcal{V}^*$. Let $\mathcal{G}$ be an accompanying graph on n^* vertices that can be either directed or undirected. The vertex connectivity of v to $\mathcal{S}$ in W is $f(v, \mathcal{S}, W) = \sum_{\{u \in \mathcal{S} \,:\, u \in \mathcal{N}(v)\}} w_{v,u}$. where $\mathcal{N}(v)$ are the (outgoing-)/neighbors of v in the directed/undirected graph $\mathcal{G}$.

We also refer to *absolute vertex connectivity* which is $|f(v, \mathcal{S}, W)|$. As we did in the initialization step, to greedily place the remaining nodes we permute $\mathcal{V}^* \setminus (\mathcal{S}_1 \cup \mathcal{S}_2)$. We place $v \in \mathcal{V}^*$ in permutation order: if $|f(v, \mathcal{S}_1, W)| \geq |f(v, \mathcal{S}_2, W)|$ then we update $\mathcal{S}_2 = \mathcal{S}_2 \cup \{v\}$. Else, we update $\mathcal{S}_1 = \mathcal{S}_1 \cup \{v\}$. Node v is placed in the set that it is less connected to in order to greedily maximize the absolute value of the connectivity across the cut. Once all nodes are placed, we rename $\mathcal{S}_1$ to $\mathcal{S}$. The details of *GRASP* greedy-construction are in Algorithm 24 using the approximation variant we explain in Section 10.2.3.

Now that we have a feasible solution $\mathcal{S}$, the local search phase looks over *neighbor cuts* of $\mathcal{S}$ to find a solution with higher absolute connectivity. The neighboring cuts of $\mathcal{S}$ are $\mathcal{S}' \subseteq \mathcal{V}^*$ one node different from $\mathcal{S}$. That is, a cut $\mathcal{S}'$ is a neighbor if and only if either (1) $\mathcal{S}' = \mathcal{S} \cup \{v\}$ for some $v \notin \mathcal{S}$ or (2) $\mathcal{S}' = \mathcal{S} \setminus \{v\}$ for some $v \in \mathcal{S}$. In each local search step, we permute $\mathcal{V}^*$ and visit them in vertex order. If $v \in \mathcal{S}$, we consider $\mathcal{S}' = \mathcal{S} \setminus \{v\}$. Else, we consider $\mathcal{S}' = \mathcal{S} \cup \{v\}$. If $|\partial(\mathcal{S}', W)| > |\partial(\mathcal{S}, W)|$, then $\mathcal{S} = \mathcal{S}'$. After all the nodes are visited, if no changes were made to $\mathcal{S}$ or we have reached a maximum number of search steps then we terminate the search. If any changes were made to $\mathcal{S}$, we repeat another local search step. The details of *GRASP* local search are in Algorithm 25 using the approximation variant we explain in Section 10.2.3.

We repeat the greedy construction and local search K times keeping track of the $\mathcal{S}$ with the largest $|\partial(\mathcal{S}', W)|$. After K, we output the $\mathcal{S}$ with the largest $|\partial(\mathcal{S}', W)|$ (Algorithm 26).

10.2.3 Approximate Grasp: Sub-neighborhood

In order to scale to larger graphs, we introduce a *GRASP* variant which looks at only a sub-set of entries of W to compute an approximation of $\partial(\mathcal{S}, W)$ during both the greedy construction and local search step of each round. Re-writing connectivity in terms of vertex-connectivity we have $\partial(\mathcal{S}, W) = \sum_{v \in \overline{\mathcal{S}}} f(v, \mathcal{S}, W)$. For each node v, instead of keeping track of all n entries of b_v where b_v is the v-th row of W, we keep track of a subset of size αn^* where $\alpha \in (0, 1)$. We call this subset the *outgoing neighbors* of v written as $\mathcal{N}_{OUT}(v)$. Now, whenever we compute vertex-connectivity of v to $\mathcal{S}$ we only use nodes in $\mathcal{N}_{OUT}(v)$ and compute $f(v, \mathcal{N}_{OUT}(v) \cap \mathcal{S}, W)$. The step where we sample a subset of neighbors for each node is called *neighborhood sub-set sampling* (Algorithm 23). neighborhood sub-set sampling outputs a directed graph $\mathcal{G}'$ where the out-degree of each node is αn^* that encodes the incoming and outgoing neighbors for each vertex.

The reason this is useful is to prevent updating the connectivity of all nodes each time we remove/add a vertex to $\mathcal{S}$ in local-search. If W is dense, the connectivity of nearly every node to $\mathcal{S}$ will change if we remove/add v from $\mathcal{S}$. With neighborhood sub-sampling, only the connectivity of nodes u such that $v \in \mathcal{N}_{OUT}(u)$ are changed as a result of removing/adding v. To keep track of which nodes to update when we add/remove v, the set of nodes to update when adding/removing v are its *incoming neighbors* $\mathcal{N}_{IN}(v)$ which is the set of nodes u with $v \in \mathcal{N}_{OUT}(u)$ the .

Most of the A we start from are non-zero almost everywhere, resulting in W non-zero everywhere.

```
Algorithm 23: Neighborhood sub-set sampling.
    Result: Directed graph $\mathcal{G}'$
    Input: Weight matrix $W$ of dimension $n^* \times n^*$ ;
    Neighborhood sample size $\alpha \in (0, 1)$ ;
    Initialize directed graph $\mathcal{G}'$ with nodes $\mathcal{V}^*$ and without edges ;
    for $v \leftarrow 1$ to $n^*$ do
        Sample node set $\mathcal{N}_{OUT}(v)$ of size $\alpha \cdot n^*$ from $\mathcal{V}^*$ without replacement where each $u \in \mathcal{V}^*$
          is sampled with probability $|w_{v,u}|/Z$ with $Z = \sum_u |w_{v,u}|$. ;
        Insert an directed edge from $v$ to $u$ for all $u \in \mathcal{N}_{OUT}(v)$ in $\mathcal{G}'$;
    end
```

However, many of the entries are near-zero so they have little effect on the connectivity of any cut. With this in mind, we sample $\mathcal{N}_{OUT}(v)$ by including vertex u with probability proportional to $w_{v,u}$ so that the vertices that impact the connectivity of v the most are with high-probability included.

10.3 Cut correction: constructing perturbation

Once we find a $\mathcal{S}$ to correct, we construct a symmetric perturbation matrix E such that $\partial(\mathcal{S}, A+E) = \partial(\mathcal{S}, A^*)$ and the entries of $A' = A + E$ satisfy $a''_{v,u} \in [0, 1]$. We construct E in two stages (1) node pair partition assignment (2) node partition update.

In the node pair partition assignment stage, we partition node pairs $\mathcal{V}^* \times \mathcal{V}^*$ into three sets $\mathcal{Q}_{\mathcal{S}} = Q_{\mathcal{S}}, Q_{\overline{\mathcal{S}}}, Q_{\mathcal{S},\overline{\mathcal{S}}}$: pairs with both, neither, or one nodes in $\mathcal{S}$.

Definition 10.3.1. Node pair partition induced by $\mathcal{S}$ Let $\mathcal{S} \subseteq \mathcal{V}^*$. Then the node pair partition $\mathcal{Q} = Q_{\mathcal{S}}, Q_{\overline{\mathcal{S}}}, Q_{\mathcal{S},\overline{\mathcal{S}}}$ where

- $Q_{\mathcal{S}} = \{(v, u) \in (\mathcal{V}^* \times \mathcal{V}^*) \ : \ v < u, v \in \mathcal{S}, u \in \mathcal{S}\}$
- $Q_{\overline{\mathcal{S}}} = \{(v, u) \in (\mathcal{V}^* \times \mathcal{V}^*) \ : \ v < u, v \notin \mathcal{S}, u \notin \mathcal{S}\}$
- $Q_{\mathcal{S},\overline{\mathcal{S}}} = \{(v, u) \in (\mathcal{V}^* \times \mathcal{V}^*) \ : \ v < u, v \in \mathcal{S}, u \notin \mathcal{S}\}$

Definition 10.3.2. Assignment by X on node pair partition induced by $\mathcal{S}$ For $\mathcal{S} \subseteq \mathcal{V}^*$ and node pair partition $\mathcal{Q}_{\mathcal{S}}$ (Definition 10.3.1), the assignment on $\mathcal{Q}_{\mathcal{S}}$ by matrix X of dimension $\mathcal{V}^* \times \mathcal{V}^*$ is a triple $T(X, \mathcal{Q}_{\mathcal{S}}) = (t(X, Q_{\mathcal{S}}), t(X, Q_{\overline{\mathcal{S}}}), t(X, Q_{\mathcal{S},\overline{\mathcal{S}}})$. For each $Q \in \mathcal{Q}_{\mathcal{S}}$, $t(X, Q) = sum_{(v,u)\in Q} x_{v,u}$ is the total weight placed on pairs in Q by X.

Algorithm 24: Approximate *GRASP* greedy-construction.

```
Result: Cut $\mathcal{S}$;
Connectivity lists to $\mathcal{S}$ and $\overline{\mathcal{S}}$ denoted $\mathbf{s}$ and $\overline{\mathbf{s}}$;
Input: Weight matrix $W$;
Directed graph $\mathcal{G}'$;
$GRASP$ initialize size $\beta$;
Initialize sets $\mathcal{S}_1 = \mathcal{S}_2 = \emptyset$. ;
Randomly place $\beta n^*$ nodes in $\mathcal{S}_1$ and $\mathcal{S}_2$ ;
Initialize $n$ dimensional connectivity lists $\mathbf{s}$ and $\overline{\mathbf{s}}$ with all zeros. ;
Permute $\mathcal{V}^*$ with permutation $\pi$ ;
for $i = 1, 2, \ldots, \beta n^*$ do
    Place $\pi(i)$ in to $\mathcal{S}_1$ or $\mathcal{S}_2$ each with probability $\frac{1}{2}$;
end
Update connectivity for all nodes to $\mathcal{S}_1$ and $\mathcal{S}_2$ ;
for $v \in \mathcal{V}^*$ do
    for $u \in \mathcal{N}_{OUT}(v)$ do
        if $u \in \mathcal{S}_1$ then
            $s_v = s_v + w_{v,u}$ ;
        else if $u \in \mathcal{S}_2$ then
            $\overline{s}_v = \overline{s}_v + w_{v,u}$ ;
    end
end
Greedily place remaining nodes in random order ;
Permute $\mathcal{V}^* \setminus (\mathcal{S}_1 \cup \mathcal{S}_2)$ with permutation $\pi$ ;
for $i \leftarrow 1$ to $(1 - \beta) \cdot n^*$ do
    $v = \pi(i)$ ;
    if $|s_v| \geq |\overline{s}_v|$ then
        $\mathcal{S}_1 = \mathcal{S}_1 \cup \{v\}$ ;
    else
        $\mathcal{S}_2 = \mathcal{S}_2 \cup \{v\}$ ;
    for $u \in \mathcal{N}_{IN}(v)$ do
        If $v \in \mathcal{S}_1$, $s_u = s_u + w_{v,u}$. ;
        Else, $\overline{s}_u = \overline{s}_u + w_{v,u}$. ;
    end
end
$\mathcal{S} = \mathcal{S}_1$ ;
```

Algorithm 25: Approximate *GRASP* local update.

```
Result: Updated $\mathcal{S}$ and connectivity lists $\mathbf{s}$ and $\overline{\mathbf{s}}$ after making local improvements to
        increase absolute connectivity ;
Input: Weight matrix $W$;
Directed graph $\mathcal{G}'$;
Cut $\mathcal{S}$, cut complement $\overline{\mathcal{S}}$;
Connectivity lists to $\mathbf{s}$ and $\overline{\mathbf{s}}$;
Number of grasp iterations $K$ ;
Initialize $x = \partial(\mathcal{S}, W)$ and initialize seen candidate lists to empty;
Length of candidate list history to detect cycles $L$ ;
for $k \leftarrow 1$ to $K$ do
    Initialize candidate list $c$ to empty. ;
    for $v \in \mathcal{V}^*$ do
        If $v \in \mathcal{S}$ and $|x + s_v - \overline{s}_v| > |x|$, then add $v$ to $c$. ;
        Else, if $v \notin \mathcal{S}$ and $|x - s_v + \overline{s}_v| > |x|$, then add $v$ to $c$. ;
    end
    Permute $c$ with permutation $\pi$ ;
    Consider local change for each candidate in $\pi$ order ;
    for $i \leftarrow 1$ to $|c|$ do
        $v = \pi(i)$ ;
        if $v \in \mathcal{S}$ and $|x + s_v - \overline{s}_v| > |x|$ then
            Remove $v$ from $\mathcal{S}$;
            $x = x + s_v - \overline{s}_v$ ;
            for $u \in \mathcal{N}_{IN}(v)$ do
                Increment $\overline{s}_v$ by $w_{v,u}$ ;
                Decrement $s_v$ by $w_{v,u}$ ;
            end
        else if $v \notin \mathcal{S}$ and $|x - s_v + \overline{s}_v| > |x|$ then
            Add $v$ to $\mathcal{S}$;
            $x = x - s_v + \overline{s}_v$ ;
            for $u \in \mathcal{N}_{IN}(v)$ do
                Increment $s_v$ by $w_{v,u}$. ;
                Decrement $\overline{s}_v$ by $w_{v,u}$. ;
            end
    end
    Detect cycles;
    if $k > 1$ then
        If $c$ in candidate lists, break ;
        Else, add $c$ to candidate list and if candidate lists is larger than max size $L$, remove
          oldest entry.
end
```

Algorithm 26: Approximate *GRASP*.

```
Result: Cut $\mathcal{S}$ with larger absolute connectivity in $W$;
Input: Weight matrix $W$;
$GRASP$ initialization size $\beta \in (0,1)$ ;
$GRASP$ neighborhood size $\alpha \in (0,1)$ ;
Number of grasp iterations $K$ ;
Candidate list history to detect cycles $L$ ;
$\mathcal{G}'$ = neighborhoodSubsample($W$, $\alpha$) ;
for $\ell \leftarrow 1$ to $L$ do
    $\mathcal{S}$', s, $\bar{\mathbf{s}}$ = greedyConstruction($W$, $\mathcal{G}'$, $\beta$) ;
    $\mathcal{S}$' = localUpdate($W$, $\mathcal{G}'$, $\mathcal{S}$', s, $\bar{\mathbf{s}}$, $K$, $L$) ;
    if $\ell = 0$ then
        best = $|\partial(\mathcal{S}', W)|$ ;
        $\mathcal{S}$ = $\mathcal{S}$' ;
    else if $|\partial(\mathcal{S}', W)| > best$ then
        best = $|\partial(\mathcal{S}', W)|$ ;
        $\mathcal{S}$ = $\mathcal{S}$' ;
end
```

Our goal is to find a feasible *assignment* on $\mathcal{Q}_{\mathcal{S}}$ by matrix $A + E$ before constructing E (Definition 10.3.2). We denote the variables for this assignment as $Y(\mathcal{Q}_{\mathcal{S}}) = (y(Q_{\mathcal{S}}), y(Q_{\overline{\mathcal{S}}}), y(Q_{\mathcal{S},\overline{\mathcal{S}}}))$. The feasibility of $Y(\mathcal{Q}_{\mathcal{S}})$ is defined by the assignment $T(A^*, \mathcal{Q}_{\mathcal{S}})$ imposed by $\mathcal{G}^*$.

We consider two ways of defining the set of feasible assignments $Y(\mathcal{Q}_{\mathcal{S}})$ subject to $T(A^*, \mathcal{Q}_{\mathcal{S}})$, both of which match the connectivity across $\mathcal{S}$. The first is the *single* constraint that $\partial(\mathcal{S}, A{+}E) = \partial(\mathcal{S}, A^*)$ which is equivalent to $y(Q_{\mathcal{S},\overline{\mathcal{S}}})) = t(A^*, Q_{\mathcal{S},\overline{\mathcal{S}}})$. This single constraint alone is enough to ensure that any matrix X with feasible $Y(\mathcal{Q}_{\mathcal{S}})$ will match the connectivity of A^*. The definition of feasible $Y(\mathcal{Q}_{\mathcal{S}})$ is the *triple* constraint which imposes two additional constraints. For the triple constraint, $y(Q_{\mathcal{S},\overline{\mathcal{S}}})) = t(A^*, Q_{\mathcal{S},\overline{\mathcal{S}}})$, $y(Q_{\mathcal{S},\overline{\mathcal{S}}})) = t(A^*, Q_{\mathcal{S},\overline{\mathcal{S}}})$, and $y(Q_{\mathcal{S},\overline{\mathcal{S}}})) = t(A^*, Q_{\mathcal{S},\overline{\mathcal{S}}})$ so there is only one feasible assignment $Y(\mathcal{Q}_{\mathcal{S}})$. For the single constraint, we optimize over the set of all feasible $Y(\mathcal{Q}_{\mathcal{S}})$ by minimizing the changes to the assignment $T(A, \mathcal{Q}_{\mathcal{S}})$. In addition to matching the connectivity across $\mathcal{S}$, we also constrain $Y(\mathcal{Q}_{\mathcal{S}})$ to be non-negative and not exceed the number of pairs in Q so that when we do construct E, the entries of $A + E$ can be within $[0, 1]$. To match the number of edges in $\mathcal{G}^*$ in expectation, we constrain $y(\mathcal{Q}_{\mathcal{S}})$ to sum up to m^*. The set of feasible assignments $Y(\mathcal{Q}_{\mathcal{S}})$ can be written as a quadratic program (Equation 10.2).

$$\begin{aligned}
&\text{Minimize} && \sum_{Q \in \mathcal{Q}_{\mathcal{S}}} (y(Q) - t(A,Q))^2 && && (10.2)\\
&\text{subject to} && y(Q_{\mathcal{S},\overline{\mathcal{S}}}) = t(A^*, Q_{\mathcal{S},\overline{\mathcal{S}}}) && &&\\
& && y(Q_{\mathcal{S},\overline{\mathcal{S}}}) \geq 0 && Q \in \mathcal{Q}_{\mathcal{S}} &&\\
& && y(Q_{\mathcal{S},\overline{\mathcal{S}}}) \leq |Q| && Q \in \mathcal{Q}_{\mathcal{S}} &&\\
& && \sum_{Q \in \mathcal{Q}_{\mathcal{S}}} y(Q_{\mathcal{S},\overline{\mathcal{S}}}) = m^* && Q \in \mathcal{Q}_{\mathcal{S}} &&
\end{aligned}$$

Now that we have a satisfying assignment for $\mathcal{Q}_{\mathcal{S}}$, we can treat each set of pairs separately. For each $Q \in \mathcal{Q}_{\mathcal{S}}$, we use the assignment $y(Q)$ to construct a perturbation matrix $E(Q)$ whose entries sum up to $y(Q) - t(A,Q)$ and $a_{v,u} + e(Q)_{v,u} \in [0,1]$. We look at two methods for constructing $E(Q)$, *uniform* and *push*. Initially, $E(Q) = 0^*_n$. If $y(Q) - t(A,Q)$ is negative, we remove positive value from entries of $E(Q)$. Conversely, $y(Q) - t(A,Q)$ is positive, we add positive value from entries of $E(Q)$. For both uniform and push, we remove/(or add) to $E(Q)_{v,u}$ for pairs $(u,v) \in Q'$ where $Q' \subseteq Q$. Subset Q' is constructed differently according to uniform and push. The amount $\delta_{Q'}$ removed/(or added) from $E(Q)_{v,u}$ for pairs $(v,u) \in Q'$ is the minimum $a_{v,u}$ or minimum $(1 - a_{v,u})$ so that $a_{v,u} + e(Q)_{v,u}$ is at least 0/at most 1. To guarantee progress, we construct Q' to be all pairs (v,u) at least/(or at most) ϵ/(or $1-\epsilon$) so $\delta_{Q'} \geq \epsilon$ when removing/(or adding). For the uniform method, this is the only constraint on Q'. For the push method, we aim to remove/(or add) from low/(or high) values in order to push lower values lower and higher values higher. This is to preserve any clustering structure that exists in A. A "median" value k is maintained that is initially equal to the median of entries $a_{v,u}$ above ϵ. The push method defines Q' to be all pairs (v,u) above/(or below) $\epsilon/1-\epsilon$ so $\delta_{Q'} \geq \epsilon$ when removing/(or adding) and below/(or above) k to remove/(or add) from relatively small/(or large) values.

For both push and uniform, $E(Q)$ is updated by removing/adding $\delta_{Q'}$ from/(to) all $e(Q)_{v,u}$ for $(v,u) \in Q'$. This procedure is repeated, updating Q' and $E(Q)$ until Q' is empty or the sum of the entries of $E(Q)$ are equal to $y(Q) - t(A,Q)$. If Q' is empty, we relax the constraints on Q' to include more pairs. If using the uniform update, we divide ϵ by 2 to relax the constraint. For push, we add/remove a constant c to k until k reaches 1 or 0. At that point, we divide ϵ by 2. The entire cut correction algorithm is in Algorithm 27.

Algorithm 27: Cut correction

```
Result: Probabilistic graph $A$ that matches the connectivity across $\mathcal{S}$ to $A^*$
Input: Probabilistic graph $A$ of dimension$n^* \times n^*$ ;
Target graph adjacency matrix $A^*$;
Cut $\mathcal{S} \subseteq \mathcal{V}^*$ ;
typeConstraint which takes value single or triple ;
typeUpdate which takes value unif or push ;
Progress parameter $\epsilon$ ;
Constant to relax feasible pair constraints $c$ ;
Compute node partition $\mathcal{Q}_\mathcal{S}$ from $\mathcal{S}$ (Definition 10.3.1) ;
Compute assignment $T(A^*, \mathcal{Q}_\mathcal{S})$ induced by $A^*$ on $\mathcal{Q}_\mathcal{S}$ that totals the weight on each
  sub-matrix defined by pairs in $\mathcal{Q}_\mathcal{S}$ ;
if typeConstraint is single then
    Compute assignment $T(A, \mathcal{Q}_\mathcal{S})$ induced by $A$ on $\mathcal{Q}_\mathcal{S}$ that totals the weight on each
      sub-matrix defined by pairs in $\mathcal{Q}_\mathcal{S}$ ;
    Compute feasible assignment $Y(\mathcal{Q}_\mathcal{S})$ that minimizes changes to $T(A, \mathcal{Q}_\mathcal{S})$ by solving
      quadratic program 10.2 ;
else
    $Y(\mathcal{Q}_\mathcal{S}) = T(A^*, \mathcal{Q}_\mathcal{S})$ ;
Construct $E$ (Algorithm 28) ;
Return $A = A + E$ ;
```

10.4 Plots

10.4.1 Trade off between entropy and matching spectra

We ran extensive experiments for the triple cut constraint with push updates. We found that across all graphs, that correcting cuts helps match spectra measured by ℓ_2^{LW}. (Figures 10.1,10.2, and 10.3). This comes at an entropy cost that is (Figures 10.4,10.5, and 10.6). The entropy decrease happens rapidly for cut corrections with a Random walk generation baseline.

Unsurprisingly, the Random walk generation baselines outperform the uniform baselines for a small number of corrected cuts (around 15 or fewer). However, the uniform baseline does perform comparably for most inputs once enough cuts have been corrected (around 30) and is much quicker to construct (Section **??**). For some inputs, the uniform baseline is inferior to the Random walk generation baselines in matching the spectrum regardless of how many cuts we correct (e.g., ROME and HEALTH).

Algorithm 28: Construct Perturbation Matrix.

```
Result: Perturbation matrix $E$
Input: Probabilistic graph $A$ of dimension$n^* \times n^*$ ;
Target graph adjacency matrix $A^*$;
Cut $\mathcal{S} \subseteq \mathcal{V}^*$ ;
typeConstraint which takes value single or triple ;
typeUpdate which takes value unif or push ;
Progress parameter $\epsilon$ ;
Constant to relax feasible pair constraints $c$ ;
$E = 0_{n^*}$ ;
for $Q \in \mathcal{Q}_{\mathcal{S}}$ do
    $E(Q)$ is the sub-matrix $E$ of entries $e(Q)_{v,u}$ of $(v,u) \in Q$ ;
    $k$ is the median of all entries in $E(Q)$ above $\epsilon$ ;
    while $\sum_{(v,u)\in Q} e(Q)_{v,u} < y(Q) - t(A,Q)$ do
        if typeUpdate is unif then
            if $y(Q) - t(A,Q) > 0$ then
                $Q' = \{(v,u) \in Q \ : \ e_{v,u} < 1-\epsilon\}$
            else
                $Q' = \{(v,u) \in Q \ : \ e_{v,u} > \epsilon\}$
        else
            if $y(Q) - t(A,Q) > 0$ then
                $Q' = \{(v,u) \in Q \ : \ e_{v,u} < 1-\epsilon \& e_{v,u} > k\}$
            else
                $Q' = \{(v,u) \in Q \ : \ e_{v,u} > \epsilon \& e_{v,u} < k\}$
        if $|Q'| = \emptyset$ then
            if typeUpdate is unif then
                $\epsilon = \frac{1}{2}\epsilon$
            else
                if $y(Q) - t(A,Q) > 0$ then
                    $k = \max(0, k-c)$ ;
                    if $k$ is 0 then
                        $\epsilon = \frac{1}{2}\epsilon$
                else
                    $k = \min(0, k+c)$ ;
                    if $k$ is 1 then
                        $\epsilon = \frac{1}{2}\epsilon$
        else
            if $y(Q) - t(A,Q) > 0$ then
                $\delta(Q') = \min(|y(Q) - t(A,Q)|/|Q'|, \min_{(v,u)\in Q'} 1 - a_{v,u})$ ;
                sign $= 1$ ;
            else
                $\delta(Q') = \min(|y(Q) - t(A,Q)|/|Q'|, \min_{(v,u)\in Q'} a_{v,u})$ ;
                sign $= -1$ ;
            for $(v,u) \in Q'$ do
                $a_{v,u} = a_{v,u} + \mathrm{sign}\delta(Q')$ ;
                $a_{u,v} = a_{u,v} + \mathrm{sign}\delta(Q')$ ;
            end
    end
end
```

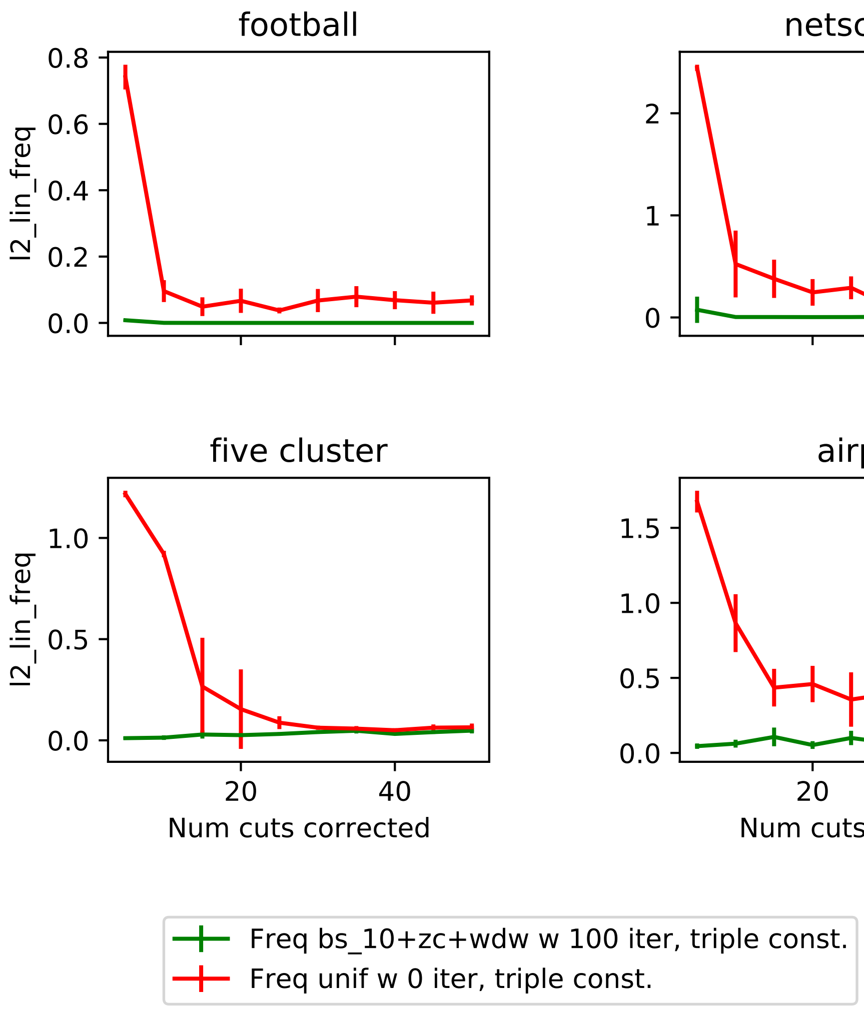


Figure 10.1: Number of cuts corrected against ℓ_2^{LW} $(\boldsymbol{\lambda}^*,\boldsymbol{\lambda})$ for Football, Netscience, Five Cluster, and Airport graphs.

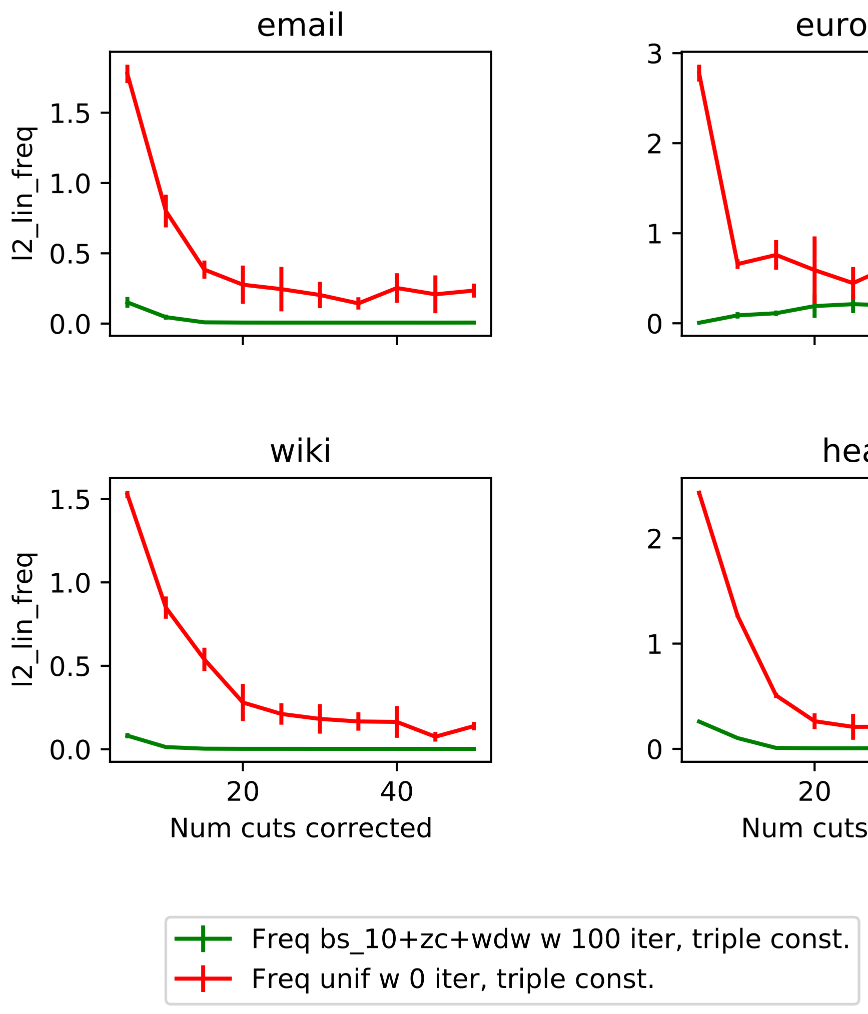


Figure 10.2: Number of cuts corrected against ℓ_2^{LW} $(\boldsymbol{\lambda}^*,\boldsymbol{\lambda})$ for Email, EuroRoad, Wiki, and Health graphs.

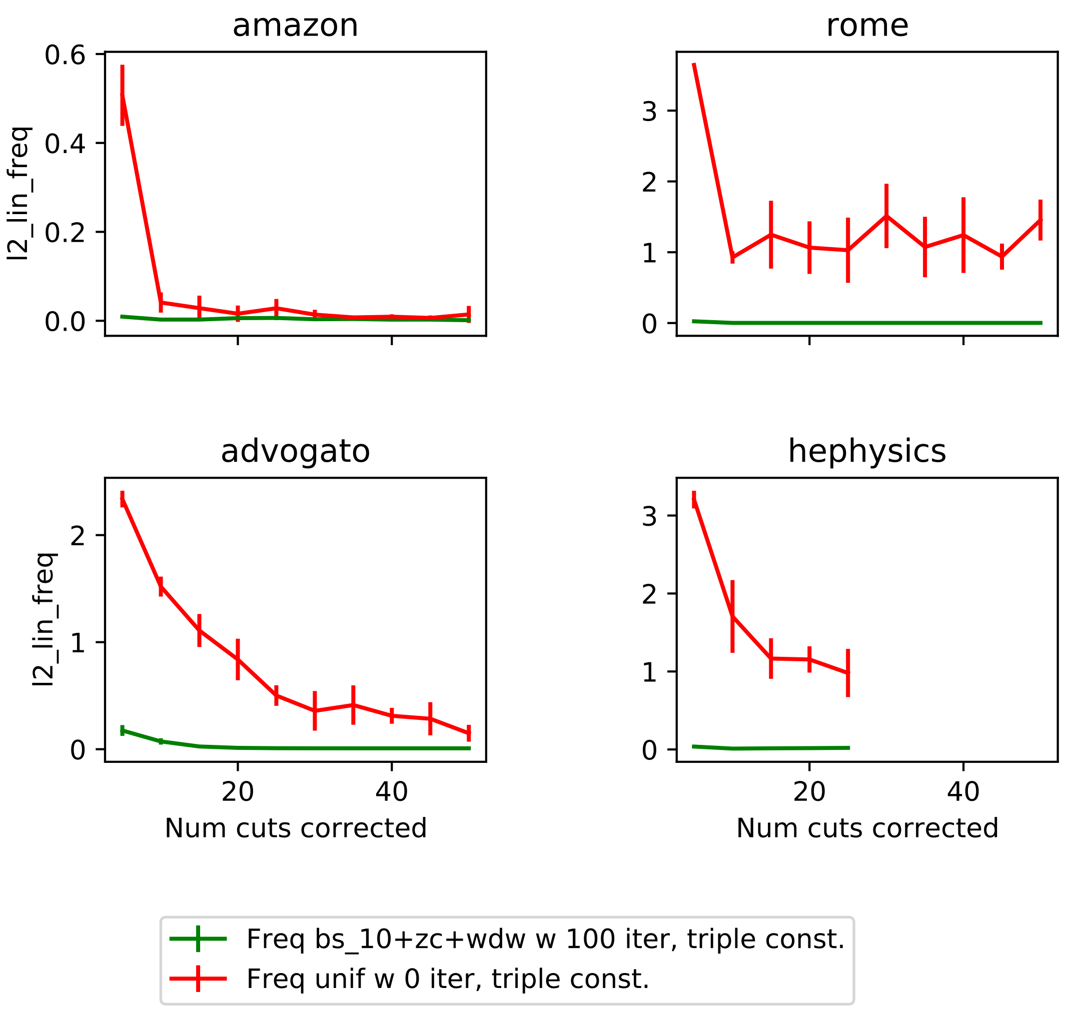


Figure 10.3: Number of cuts corrected against ℓ_2^{LW} $(\boldsymbol{\lambda}^*,\boldsymbol{\lambda})$ for Amazon, Rome, Advogato, and HEPhysics graphs.

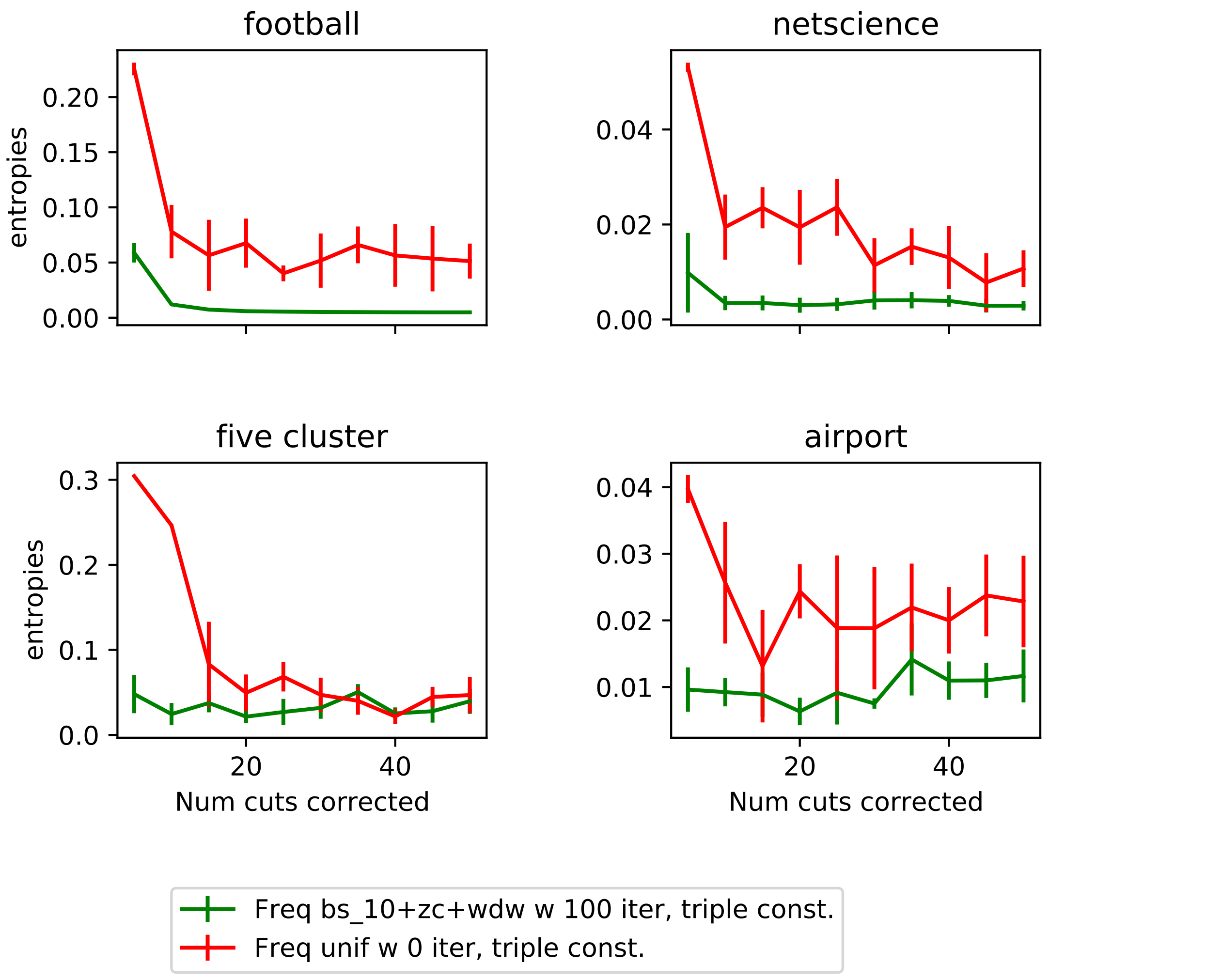


Figure 10.4: Number of cuts corrected against mean entropy $h(a_{u,v})$ for FOOTBALL, NETSCIENCE, FIVE CLUSTER, and AIRPORT graphs.

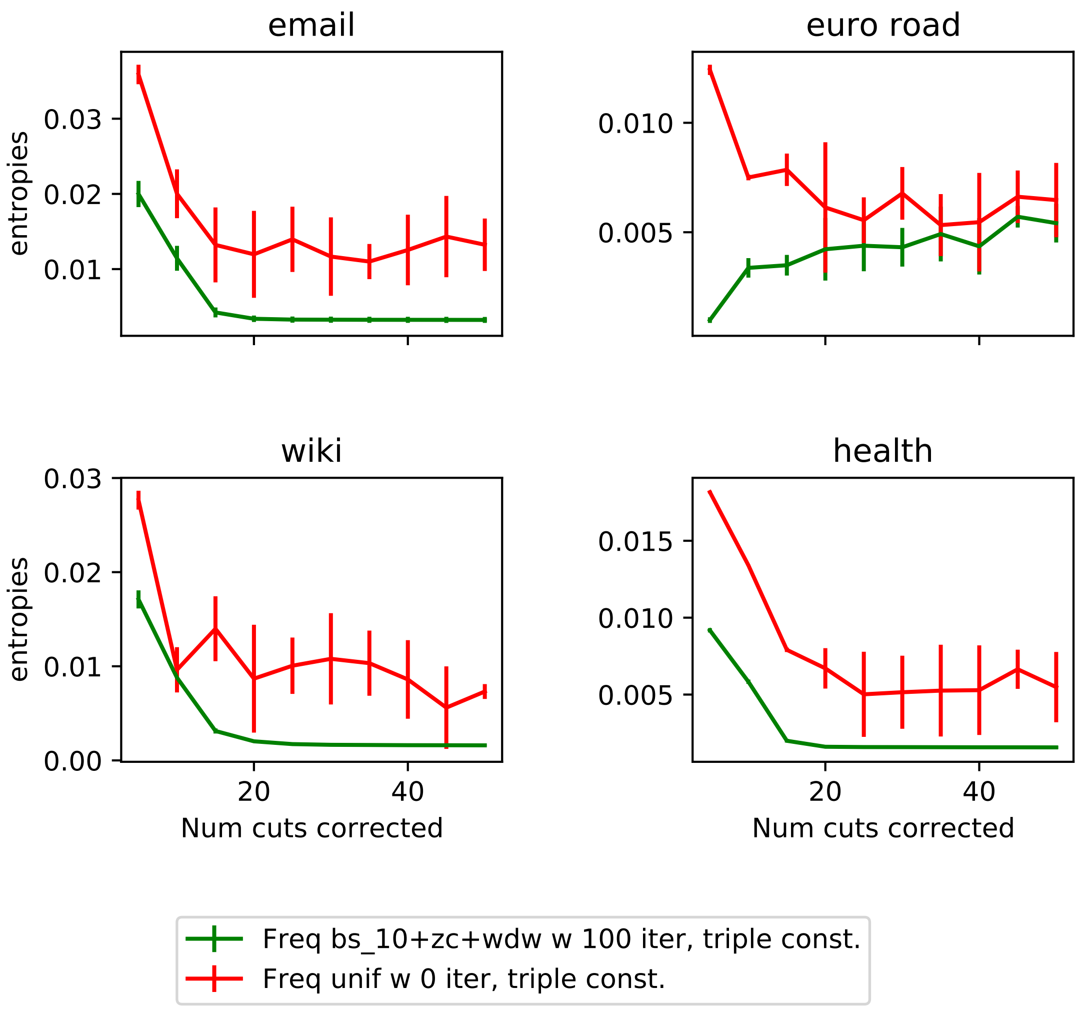


Figure 10.5: Number of cuts corrected against mean entropy $h(a_{u,v})$ for EMAIL, EUROROAD, WIKI, and HEALTH graphs.

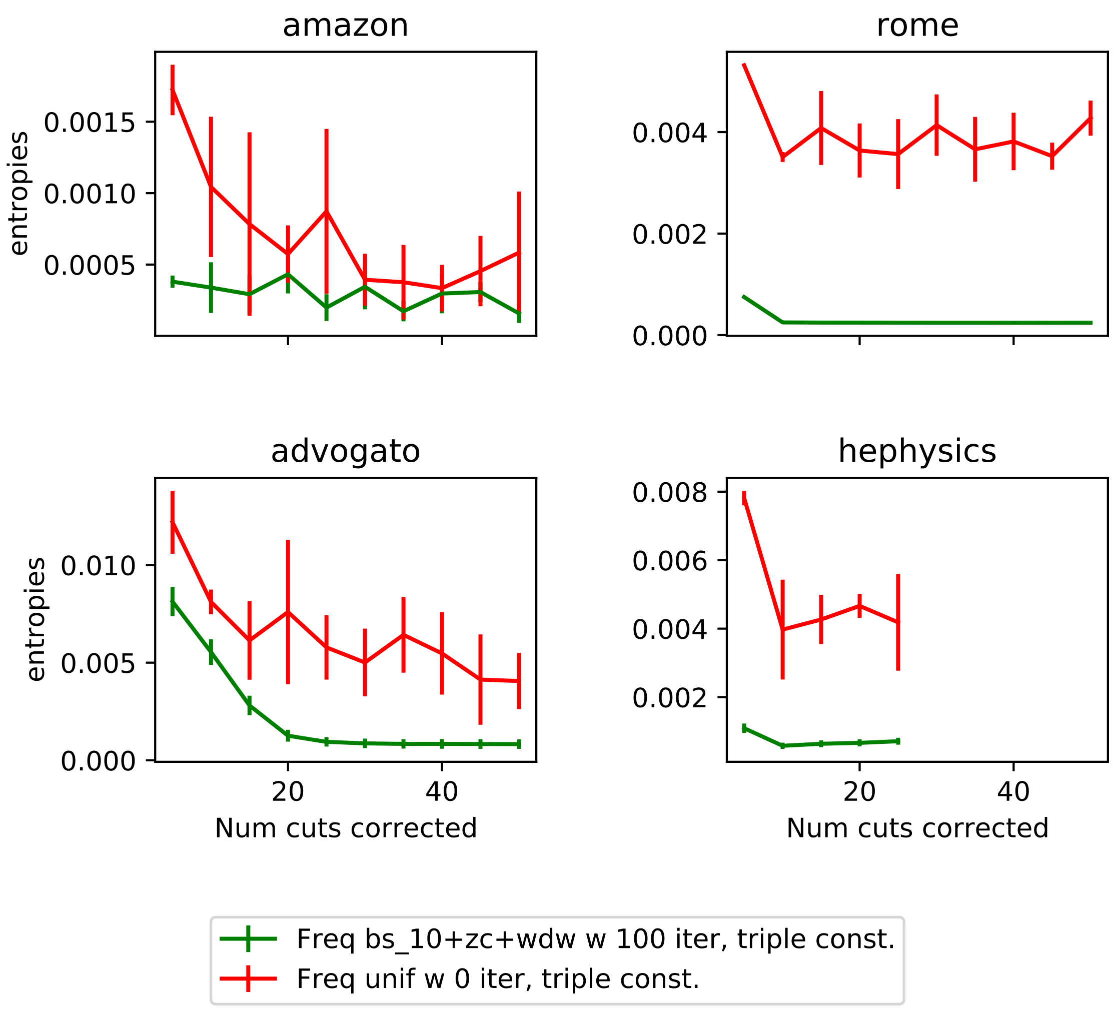


Figure 10.6: Number of cuts corrected against mean entropy $h(a_{u,v})$ for Amazon, Rome, Advogato, and HEPhysics graphs.

Graph	Neighborhood size .5	Neighborhood size 1.0
Airport	$1.8 \pm .12$	$1.34 \pm .1$
Email	$6.26 \pm .41$	$7.74 \pm .45$
Rome	63.1 ± 7	74.6 ± 6.12
Advogato	181.1 ± 21.7	239.1 ± 15.3

Table 10.1: Time it takes to sample and correct a cut using approximate *GRASP* (Algorithm 26). Means taken from sampling and correcting 50 cuts from Random walk generation baseline.

10.4.2 Alternative cut correction algorithms

We saw that uniform updates do not reliably decrease the ℓ_2^{LW} norm on the spectra as the push updates do (Figure **??**). While the uniform updates to improve spectral performance on the uniform baseline probababilistic adjacency matrices, the improvements are much slower than with the push updates. The single constraint performance is comparable, but the triple constraints more reliably improve the spectral performance with each cut corrected on the Random walk generation baseline probababilistic adjacency matrices.

10.4.3 Time of Cut fix generation compared to Random walk generation

Figure 10.8 compares the time it takes to sample and correct each cut on the two different baseline probababilistic adjacency matrices (uniform and Random walk generation). We see that the time to correct the cuts sampled using both baselines is comparable. The time it takes to generate a baseline using Random walk generation is considerable, although we do see in improvement in spectral performance (Section 10.4.1).

It takes under 17 hours to sample a baseline Random walk generation probababilistic adjacency matrix with 50 random walk iterations and correct 25 cuts for our largest input graph HEPhysics. This is a significant improvement over the NetGAN approach, which can not make significant progress on the Health graph within 24 hours.

10.4.4 Time saved using neighborhood sub-sample

We observe that limiting the grasp neighborhood to .5 does help the time it takes to sample and correct a cut on average as the graph size grows. However, the gains are not significant.

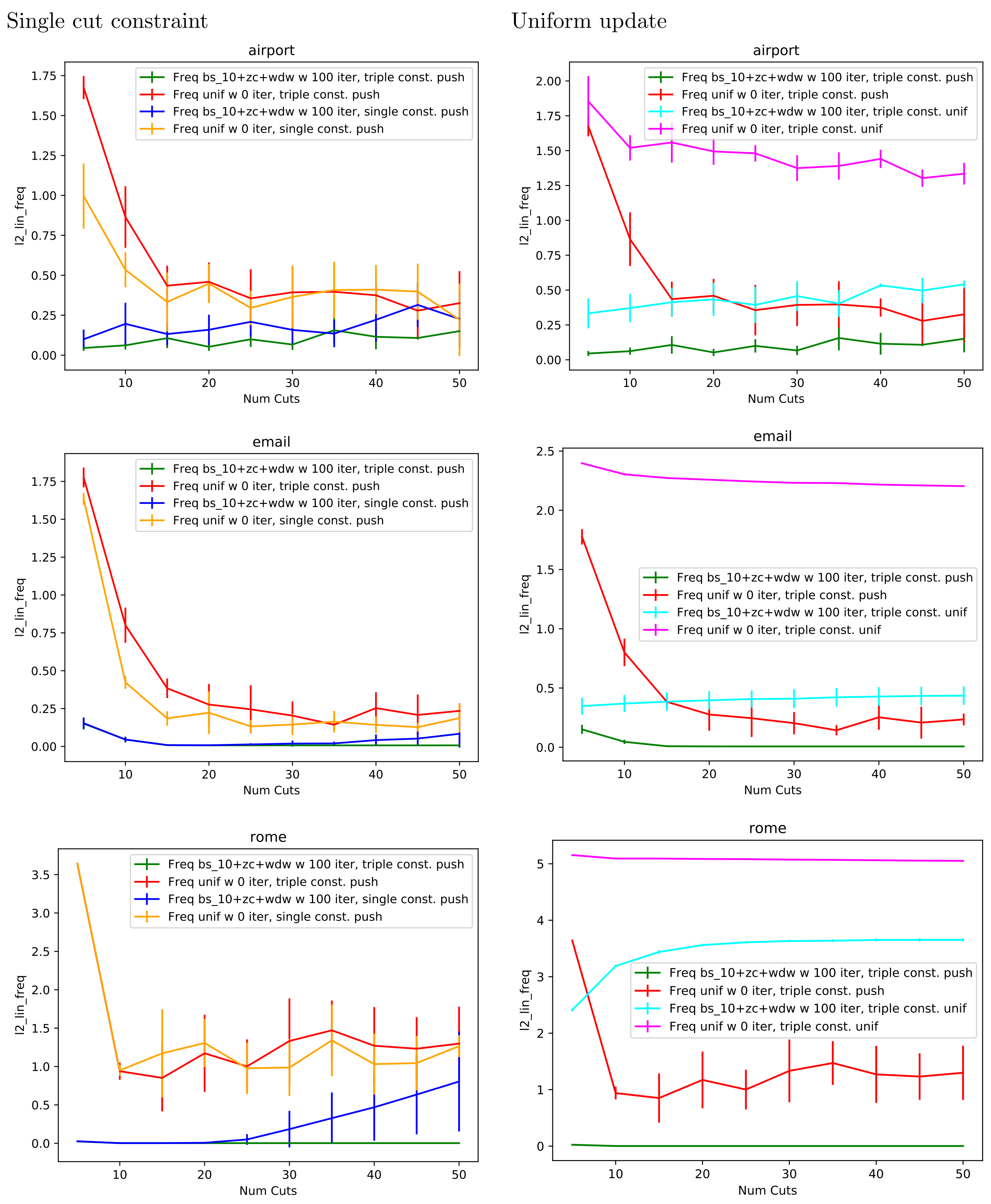


Figure 10.7: Comparing the spectral performance of the different constraints (single and triple) and different updates (uniform and push). The triple constraint seems to be more stable than single, while the push updates improve performance far faster and more reliably than uniform.

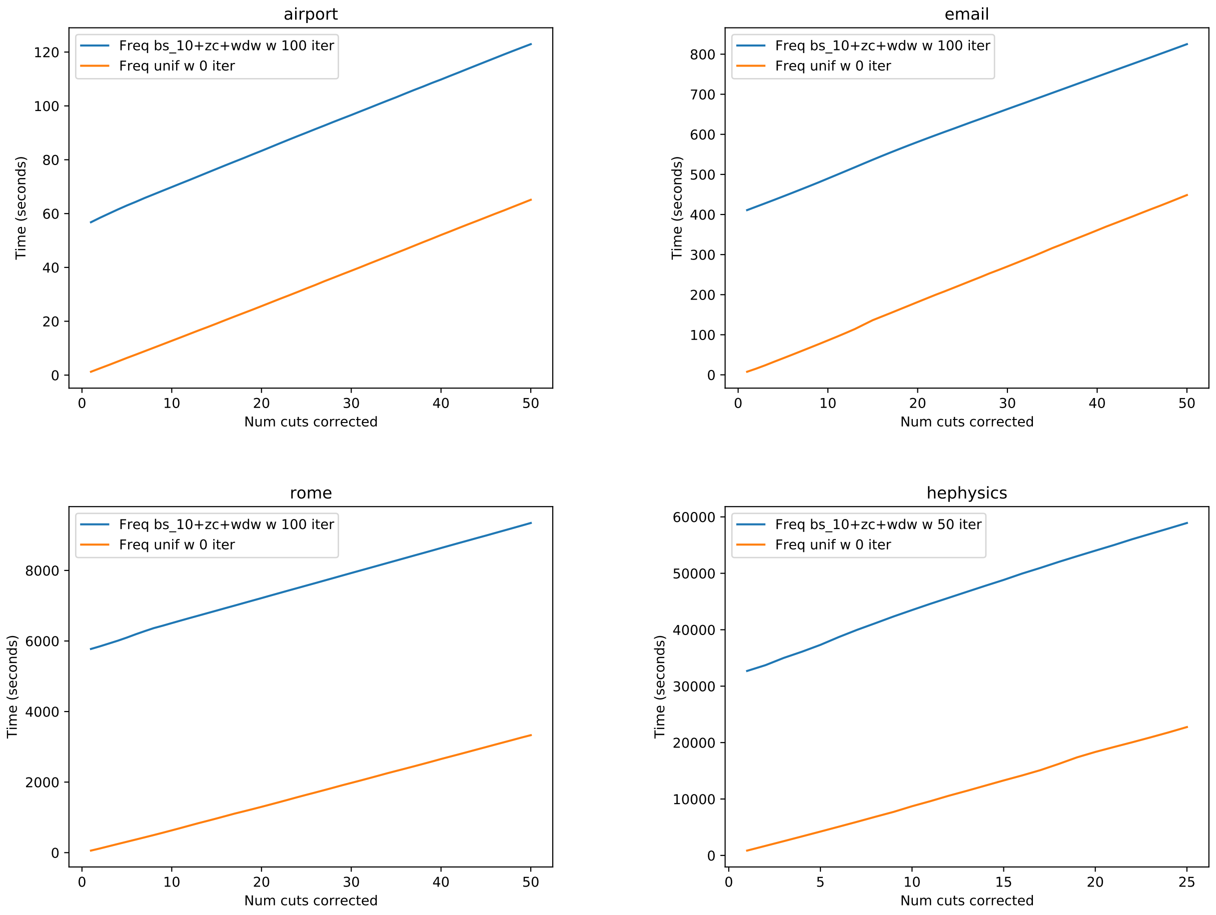


Figure 10.8: Time of correcting the cuts in a uniform baseline probababilistic adjacency matrix compared with fixing a Random walk generation probababilistic adjacency matrix, including the time it takes to construct the baseline. The time it takes to construct the Random walk generation probababilistic adjacency matrix is significant.

Chapter 11

Results

In this chapter we provide a comparison across the models introduced in this thesis and three benchmark models on a subset of target graphs. We compare the models on how well they achieve both the diversity objective (measured using entropy of bernoulli edge probabilities) and similarity objective (measured by comparing graph statistics of generated graphs to target graphs) (Chapter 3). The models and hyperparamaters used along with abbreviations are listed in Table 11.1. The target graphs used with abbreviations are listed in Table 11.2.

For the Cut fix generation models, we chose these number of cuts to fix because Figures 10.1 and 10.4 show that beyond 10 cuts for baseline models generated using Random walk generation and 50 cuts for the uniform baseline the spectral performance stops improving and we correct as few cuts as possible to preserve entropy. Also for all Cut fix generation models, we use the push construction with triple constraint because it was more stable in improving matching the spectrum as we corrected more cuts in our experiments (Figure 10.7).

11.1 Diversity: comparing entropy across models

For most of the models in Table 11.1, the model prescribes probabilities that each edge is include. For these, we measure diversity by computing the average entropy of these edge probabilities (Table 11.3).

We find that the global generative models (NetGAN, Random walk generation) have higher entropy than the benchmarks (SBM and Kron) for the models that are curtailed earlier. For NetGAN, the spectral stopping criterion with edge prediction adds more training iterations which results in

Model abbreviation	Model
Walk/0	Random walk generation
Walk/5	Random walk generation with 5 cuts corrected using Cut fix generation.
Walk/10	Random walk generation with 10 cuts corrected using Cut fix generation.
Unif/10	Cut fix generation on the *uniform* baseline matrix with uniform edge probabilities with 10 cuts corrected using Cut fix generation.
Unif/50	Cut fix generation on the *uniform* baseline matrix with uniform edge probabilities with 50 cuts corrected using Cut fix generation. that sum up to m^*.
NG/S/EP	NetGAN trained with the standard random walk until the edge-prediction stopping criterion is met.
NG/C/EP	NetGAN trained with the combination random walk until the edge-prediciton stopping criterion is met.
NG/S/SE	NetGAN trained with the standard random walk until the spectral stopping criterion with edge prediction is met.
NG/C/SE	NetGAN trained with the combination random walk until the spectral edge-prediction stopping criterion is met.
SpecGen	Spectral generation
Config	Configuration graph model
Kron	Kronecker
SBM	Stochastic Block Model

Table 11.1: Models and their abbreivations that are compared in this chapter along diversity and similarity metrics. Models introduced by this thesis and models trained with variants introduced in this thesis are bolded.

the entropy dropping below the benchmarks for some models (Football with combination walk, Airport with standard randomw walk). For Cut fix generation, we see that as we fix more cuts the entropy drops (Figure 10.4 and 10.5). For both the uniform baseline with 50 cuts corrected and the Random walk generation baseline with 5 cuts corrected, the entropy is below the benchmarks.

Target graph abv.	Target graph
Football	Football
Airport	Airport
Netscience	NetSci
Five Cluster	Five Cl.
Email	Email

Table 11.2: Target graphs used for the benchmark comparison and their abbreviations. A table of the properties of the input graphs is in Table 6.1.

Graph	Walk/0	Walk/5	Unif/10	Unif/50	NG/S/EP	NG/S/SE	NG/C/EP	NG/C/SE	Kron	SBM
Football	0.15 ± 0.01	0.05 ± 0.01	0.09 ± 0.02	0.05 ± 0.02	0.17 ± 0.02	0.13 ± 0.04	**0.22** ± **0.02**	0.04 ± 0.0	0.16	0.14
NetSci	**0.03** ± **0.0**	0.01 ± 0.01	0.02 ± 0.01	0.01 ± 0.0	0.02 ± 0.01	0.01 ± 0.0	0.02 ± 0.01	0.01 ± 0.0	0.02	0.03
Airport	0.04 ± 0.0	0.01 ± 0.0	0.03 ± 0.01	0.02 ± 0.01	0.04 ± 0.0	0.03 ± 0.01	**0.07** ± **0.01**	0.06 ± 0.01	0.05	0.05
Five Cl.	0.12 ± 0.01	0.05 ± 0.02	**0.25** ± **0.0**	0.05 ± 0.02	0.14 ± 0.01	0.13 ± 0.0	0.15 ± 0.02	0.13 ± 0.0	0.19	0.07
Email	**0.03** ± **0.0**	0.02 ± 0.0	0.02 ± 0.0	0.01 ± 0.01	**0.03** ± **0.0**	0.02 ± 0.01	**0.03** ± **0.0**	**0.03** ± **0.0**	0.01	0.02

Table 11.3: Mean entropy of entries in the probababilistic adjacency matrices for each model. The walk algorithm combined with correcting cuts results has low entropy compared to the other global approaches and the benchmarks for most graphs (EMAIL is an exception).

11.2 Discrete across global feature methods

We perform a comparison across hyperparamaters for Random walk generation, Cut fix generation, NetGAN and Spectral generation. We find that Cut fix generation combined with Random walk generation has the best results (Walk/5 and Walk/10). The high performance of Cut fix generation and Random wak generation on generating graphs similar to the inputs comes at a diversity cost as described in Section 11.1. There are some exceptions:

- For matching the spectrum measured by ℓ_2^{LW}, NetGAN with the combination walk and spectral stopping criterion with edge prediction peforms the best on the FIVE CLUSTER graph. The combination walk was designed to discover sparse cuts like those in the FIVE CLUSTER graph, so this is perhaps unsurprising. Cut fix generation does not help to matching the spectrum with the Random walk generation baseline on the FIVE CLUSTER graph, suggesting that this method does not work well in the presence of extreemely sparse cuts as thre is in the FIVE CLUSTER graph. This is likely because the Random walk generation already has discovered the clusters in the FIVE CLUSTER graph. This causes the noise added by correcting cuts to be largely displaced because the cuts being corrected were mostly correct and adding the uniform updates destroys more structure learned by Random walk generation then is encoded by the cut being corrected.

- Spectral generation generally matches the spectral gap better than the other approaches, likely do to the aggresive rounding across the critical cuts in Algorithm 2. However, like NetGAN with the standard random walk, Spectral generation is susceptible to disconnecting the FIVE CLUSTER graph so it lagged behind the Random walk generation for matching the Spectral gap after breaking off clusters.

- The method that had the largest connected componenet varied. For all of the global feature methods, nodes can only be broken off (not added) so the goal is to keep as many nodes attached as possible. For most inputs, correcting cuts from the Random walk generation baselines helped attach nodes because cuts that are too sparse in the baseline are corrected so the nodes are reattached. However, for the FIVE CLUSTER graph, we see that correcting cuts breaks off nodes. This is likely because the *GRASP* optimizaiton finds a cluster that needs addition noise, but by appplying subtraction noise on either side of the cut breaks off nodes.

Graph	Walk/0	Walk/5	Walk/10	Unif/50	NG/S/EP	NG/S/SE	NG/C/EP	NG/C/SE	SpecG
Football	0.02 ± 0.0	0.0 ± 0.0	**0.0 ± 0.0**	0.04 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	0.06 ± 0.0	0.01 ± 0.0	0.06 ± 0.0
NetSci	0.06 ± 0.0	0.04 ± 0.0	**0.0 ± 0.0**	0.06 ± 0.0	0.04 ± 0.0	0.02 ± 0.0	0.02 ± 0.0	0.01 ± 0.0	0.04 ± 0.0
Five Cl.	0.02 ± 0.0	0.04 ± 0.0	0.05 ± 0.1	0.07 ± 0.0	0.03 ± 0.0	0.11 ± 0.1	0.01 ± 0.0	**0.01 ± 0.0**	0.34 ± 0.1
Airport	0.09 ± 0.0	0.03 ± 0.0	**0.02 ± 0.0**	0.08 ± 0.0	0.13 ± 0.1	0.09 ± 0.1	0.16 ± 0.0	0.1 ± 0.1	0.08 ± 0.0
Email	0.03 ± 0.0	0.02 ± 0.0	**0.01 ± 0.0**	0.04 ± 0.0	0.04 ± 0.0	0.02 ± 0.0	0.03 ± 0.0	0.02 ± 0.0	0.02 ± 0.0

Table 11.4: Average difference in spectrum measured by ℓ_2^{LW} $(\boldsymbol{\lambda}^*, \boldsymbol{\lambda})$ between target graphs and random graphs generated by global generative graph models. Comparison is across different hyperparamaters for the global generative graph models.

Graph	Walk/0	Walk/5	Walk/10	Unif/50	NG/S/EP	NG/S/SE	NG/C/EP	NG/C/SE	SpecG
Football	0.07 ± 0.0	0.01 ± 0.0	**0.0 ± 0.0**	0.13 ± 0.0	0.04 ± 0.0	0.02 ± 0.0	0.12 ± 0.0	0.04 ± 0.0	0.02 ± 0.0
NetSci	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	0.05 ± 0.0	0.02 ± 0.0	0.01 ± 0.0	0.02 ± 0.0	0.01 ± 0.0	**0.0 ± 0.0**
Five Cl.	**0.0 ± 0.0**	0.01 ± 0.0	0.03 ± 0.0	0.17 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.01 ± 0.0	0.0 ± 0.0	0.17 ± 0.3
Airport	0.06 ± 0.0	0.05 ± 0.0	0.07 ± 0.0	0.12 ± 0.0	0.12 ± 0.1	0.05 ± 0.0	0.13 ± 0.1	0.08 ± 0.0	**0.01 ± 0.0**
Email	0.1 ± 0.0	0.07 ± 0.0	0.03 ± 0.0	0.1 ± 0.0	0.05 ± 0.0	0.03 ± 0.0	0.06 ± 0.0	0.04 ± 0.0	**0.01 ± 0.0**

Table 11.5: Average difference in spectral gap measured by $|\lambda_2^* - \lambda_2|$ between target graphs and random graphs generated by global generative graph models. Comparison is across different hyperparamaters for the global generative graph models.

Graph	Walk/0	Walk/5	Walk/10	Unif/50	NG/S/EP	NG/S/SE	NG/C/EP	NG/C/SE	SpecG
Football	0.11 ± 0.0	0.03 ± 0.0	**0.01 ± 0.0**	0.14 ± 0.0	0.06 ± 0.0	0.04 ± 0.0	0.15 ± 0.0	0.04 ± 0.0	0.08 ± 0.0
NetSci	1.14 ± 0.4	0.99 ± 0.4	0.94 ± 0.3	1.72 ± 0.1	1.48 ± 0.3	1.26 ± 0.2	1.23 ± 0.3	1.05 ± 0.3	**0.62 ± 0.4**
Five Cl.	1.93 ± 0.1	**1.86 ± 0.8**	2.8 ± 0.7	3.54 ± 0.1	2.53 ± 0.4	2.02 ± 0.4	2.83 ± 0.4	2.5 ± 0.4	2.93 ± 0.6
Airport	0.25 ± 0.0	0.16 ± 0.0	0.13 ± 0.0	**0.12 ± 0.1**	0.25 ± 0.1	0.18 ± 0.1	0.15 ± 0.0	0.12 ± 0.0	0.15 ± 0.1
Email	0.23 ± 0.0	0.2 ± 0.0	**0.11 ± 0.0**	0.24 ± 0.0	0.26 ± 0.0	0.2 ± 0.1	0.19 ± 0.0	0.13 ± 0.0	0.17 ± 0.0

Table 11.6: Average difference in shortest path length distribution measured by Earth Mover's Distance between target graphs and random graphs generated by global generative graph models. Comparison is across different hyperparamaters for the global generative graph models.

Graph	Walk/0	Walk/5	Walk/10	Unif/50	NG/S/EP	NG/S/SE	NG/C/EP	NG/C/SE	SpecG
Football	0.13 ± 0.0	0.03 ± 0.0	**0.01 ± 0.0**	0.17 ± 0.0	0.08 ± 0.0	0.06 ± 0.0	0.2 ± 0.0	0.08 ± 0.0	0.19 ± 0.0
NetSci	0.51 ± 0.0	0.2 ± 0.1	**0.11 ± 0.0**	0.54 ± 0.0	0.32 ± 0.1	0.25 ± 0.0	0.3 ± 0.1	0.23 ± 0.0	0.49 ± 0.0
Five Cl.	0.13 ± 0.1	0.07 ± 0.0	0.06 ± 0.0	0.2 ± 0.0	0.05 ± 0.0	0.05 ± 0.0	0.07 ± 0.0	**0.04 ± 0.0**	0.27 ± 0.0
Airport	0.22 ± 0.0	**0.11 ± 0.0**	0.13 ± 0.0	0.36 ± 0.0	0.21 ± 0.0	0.17 ± 0.0	0.41 ± 0.0	0.34 ± 0.1	0.33 ± 0.0
Email	0.1 ± 0.0	0.07 ± 0.0	**0.04 ± 0.0**	0.18 ± 0.0	0.12 ± 0.0	0.06 ± 0.0	0.14 ± 0.0	0.11 ± 0.0	0.14 ± 0.0

Table 11.7: Average difference in clustering coefficient distribution measured by Earth Mover's Distance between target graphs and random graphs generated by global generative graph models. Comparison is across different hyperparamaters for the global generative graph models.

Graph	Walk/0	Walk/5	Walk/10	Unif/50	NG/S/EP	NG/S/SE	NG/C/EP	NG/C/SE	SpecG
Football	0.0 ± 0.0	**0.0 ± 0.0**	**0.0 ± 0.0**	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	**0.0 ± 0.0**	0.0 ± 0.0
NetSci	0.01 ± 0.0	**0.0 ± 0.0**	**0.01 ± 0.0**	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0
Five Cl.	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	**0.01 ± 0.0**	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0
Airport	**0.0 ± 0.0**	**0.0 ± 0.0**	**0.0 ± 0.0**	0.0 ± 0.0	**0.0 ± 0.0**	**0.0 ± 0.0**	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Email	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0

Table 11.8: Average difference in betweenness centrality distribution measured by Earth Mover's Distance between target graphs and random graphs generated by global generative graph models. Comparison is across different hyperparamaters for the global generative graph models.

Graph	Walk/0	Walk/5	Walk/10	Unif/50	NG/S/EP	NG/S/SE	NG/C/EP	NG/C/SE	SpecG
Football	1.73 ± 0.1	0.89 ± 0.1	**0.23 ± 0.0**	2.36 ± 0.4	1.84 ± 0.1	1.66 ± 0.2	1.83 ± 0.1	1.05 ± 0.1	2.52 ± 0.3
NetSci	0.75 ± 0.2	0.35 ± 0.1	**0.15 ± 0.0**	0.55 ± 0.2	0.44 ± 0.2	0.27 ± 0.1	0.65 ± 0.1	0.49 ± 0.1	0.88 ± 0.2
Five Cl.	10.67 ± 5.8	7.09 ± 0.8	5.71 ± 1.4	**3.22 ± 0.8**	5.28 ± 1.4	5.94 ± 0.5	5.68 ± 1.3	5.47 ± 0.6	13.03 ± 4.1
Airport	3.41 ± 0.2	1.16 ± 0.4	**0.87 ± 0.2**	2.33 ± 0.4	1.36 ± 0.3	1.15 ± 0.4	4.73 ± 0.4	4.2 ± 1.1	2.52 ± 0.1
Email	0.7 ± 0.1	0.4 ± 0.0	**0.23 ± 0.0**	1.08 ± 0.5	0.82 ± 0.1	0.65 ± 0.2	1.82 ± 0.1	1.84 ± 0.2	0.57 ± 0.2

Table 11.9: Average difference in degree distribution measured by Earth Mover's Distance between target graphs and random graphs generated by global generative graph models. Comparison is across different hyperparamaters for the global generative graph models.

Graph	Walk/0	Walk/5	Walk/10	Unif/50	NG/S/EP	NG/S/SE	NG/C/EP	NG/C/SE	SpecG
Football	115 ± 0	115 ± 0	115 ± 0	115 ± 0	115 ± 0	115 ± 0	115 ± 0	115 ± 0	115 ± 0
NetSci	339 ± 12	346 ± 24	**375 ± 4**	364 ± 9	355 ± 9	367 ± 9	354 ± 10	362 ± 9	330 ± 25
Five Cl.	**500 ± 0**	473 ± 30	476 ± 48	**500 ± 0**	483 ± 33	428 ± 83	497 ± 8	497 ± 7	436 ± 98
Airport	494 ± 2	496 ± 2	**497 ± 2**	474 ± 10	462 ± 15	476 ± 18	482 ± 5	479 ± 11	487 ± 5
Email	1113 ± 4	1109 ± 7	1126 ± 3	1108 ± 12	1045 ± 12	1068 ± 26	1112 ± 3	1117 ± 6	**1128 ± 7**

Table 11.10: Average size of largest connected component of graphs generated by global generative graph models. Comparison is across different hyperparamaters for the global generative graph models.

11.3 Discrete benchmark comparison

In the tables below, we compare the global feature methods to three benchmarks (1) the Congiguration model (Config) (2) the Stochastic Block model (SBM) and (3) the Kronecker model (Kron). We choose the following hyper-paramaters for our global feature methods using the entropy of each model (entropies in Table 11.3):

1. For Random walk generation, we compare without cut corrections because once cuts are corrected the entropy was much lower than SBM and Kron. Becuase we suspect that lower-entropy models perform better on our graph similarity metrics, to make the comparison more fare we use Walk/0.

2. For Cut fix generation, we compare using the Uniform baseline and 10 cuts corrections. We saw above the Uniform baseline lags behing the Random walk generation baseline for most graph similarity metrics even as the number of cut corrections reaches 50. However, we are interested in whehter Cut fix generation with the Uniform baseline can compete agains the benchmarks which have comparable entropy.

3. For NetGAN, we used the spectral edge-prediction stopping criterion because for most of the inputs the entropy remained comparable to SBM and Kron and the graph similarity performance improves compared to using the edge-prediction stopping criterion only.

For all graph similarity metrics besides degree distribution and the size of the largest connected componenet, the global feautre methods do better (spectrum ℓ_2^{LW}, spectral gap absolute value, clustering coefficient) or the same (betweenness). The NetGAN tends to do the best, with the exception of the spectral gap for which Spectral generation is superior.

For the size of the largest connected component, the Configuration and Stochatic Block models do the best job. The Kronecker model noticably lags behind because its probabilistic graphs are limited to sizes that are powers of 2.

Graph	Walk/0	Unif/10	NG/S/SE	NG/C/SE	SpecG	Config	SBM	Kron
Football	0.02 ± 0.0	0.06 ± 0.0	**0.01 ± 0.0**	0.01 ± 0.0	0.06 ± 0.0	0.16 ± 0.0	0.12 ± 0.0	0.15 ± 0.0
NetSci	0.06 ± 0.0	0.15 ± 0.0	0.02 ± 0.0	**0.01 ± 0.0**	0.04 ± 0.0	0.15 ± 0.0	0.15 ± 0.0	0.13 ± 0.0
Airport	0.09 ± 0.0	0.15 ± 0.1	0.09 ± 0.1	0.1 ± 0.1	**0.08 ± 0.0**	0.17 ± 0.0	0.45 ± 0.0	0.39 ± 0.0
Five Cl.	0.02 ± 0.0	0.69 ± 0.0	0.11 ± 0.1	**0.01 ± 0.0**	0.34 ± 0.1	0.7 ± 0.0	0.3 ± 0.2	0.7 ± 0.0
Email	0.03 ± 0.0	0.23 ± 0.0	**0.02 ± 0.0**	**0.02 ± 0.0**	**0.02 ± 0.0**	0.07 ± 0.0	0.1 ± 0.0	0.15 ± 0.0

Table 11.11: Average difference in spectrum measured by ℓ_2^{LW} $(\boldsymbol{\lambda}^*, \boldsymbol{\lambda})$ between target graphs and random graphs generated by generative graph models. Comparison between global generative graph models and bechmark models that have comparable entropy.

Graph	Walk/0	Unif/10	NG/S/SE	NG/C/SE	SpecG	Config	SBM	Kron
Football	0.07 ± 0.0	0.08 ± 0.0	0.02 ± 0.0	0.04 ± 0.0	**0.02 ± 0.0**	0.31 ± 0.0	0.06 ± 0.0	0.29 ± 0.0
NetSci	0.01 ± 0.0	0.07 ± 0.1	0.01 ± 0.0	0.01 ± 0.0	**0.0 ± 0.0**	0.15 ± 0.0	0.16 ± 0.0	0.11 ± 0.0
Airport	0.06 ± 0.0	0.08 ± 0.1	0.05 ± 0.0	0.08 ± 0.0	**0.01 ± 0.0**	0.21 ± 0.1	0.43 ± 0.0	0.37 ± 0.1
Five Cl.	**0.0 ± 0.0**	0.7 ± 0.0	**0.0 ± 0.0**	0.0 ± 0.0	0.17 ± 0.3	0.73 ± 0.0	0.02 ± 0.1	0.73 ± 0.0
Email	0.1 ± 0.0	0.18 ± 0.0	0.03 ± 0.0	0.04 ± 0.0	**0.01 ± 0.0**	0.11 ± 0.0	0.21 ± 0.0	0.05 ± 0.0

Table 11.12: Average difference in spectral gap measured by $|\lambda_2^* - \lambda_2|$ between target graphs and random graphs generated by generative graph models. Comparison between global generative graph models and bechmark models that have comparable entropy.

Graph	Walk/0	Unif/10	NG/S/SE	NG/C/SE	SpecG	Config	SBM	Kron
Football	0.11 ± 0.0	0.12 ± 0.0	**0.04 ± 0.0**	**0.04 ± 0.0**	0.08 ± 0.0	0.24 ± 0.0	0.18 ± 0.0	0.16 ± 0.0
NetSci	1.14 ± 0.4	1.75 ± 0.4	1.26 ± 0.2	1.05 ± 0.3	**0.62 ± 0.4**	2.27 ± 0.0	2.03 ± 0.1	2.0 ± 0.1
Airport	0.25 ± 0.0	0.19 ± 0.1	0.18 ± 0.1	0.12 ± 0.0	**0.15 ± 0.1**	0.18 ± 0.0	0.48 ± 0.0	0.3 ± 0.0
Five Cl.	**1.93 ± 0.1**	3.67 ± 0.0	2.02 ± 0.4	2.5 ± 0.4	2.93 ± 0.6	3.72 ± 0.0	1.94 ± 1.0	3.72 ± 0.0
Email	0.23 ± 0.0	0.53 ± 0.0	0.2 ± 0.1	**0.13 ± 0.0**	0.17 ± 0.0	0.28 ± 0.0	0.23 ± 0.0	0.96 ± 0.1

Table 11.13: Average difference in shortest path length distribution measured by Earth Mover's Distance between target graphs and random graphs generated by generative graph models. Comparison between global generative graph models and bechmark models that have comparable entropy.

Graph	Walk/0	Unif/10	NG/S/SE	NG/C/SE	SpecG	Config	SBM	Kron
Football	0.13 ± 0.0	0.24 ± 0.0	**0.06 ± 0.0**	0.08 ± 0.0	0.19 ± 0.0	0.33 ± 0.0	0.27 ± 0.0	0.32 ± 0.0
NetSci	0.51 ± 0.0	0.73 ± 0.0	0.25 ± 0.0	**0.23 ± 0.0**	0.49 ± 0.0	0.72 ± 0.0	0.72 ± 0.0	0.72 ± 0.0
Airport	0.22 ± 0.0	0.24 ± 0.1	**0.17 ± 0.0**	0.34 ± 0.1	0.33 ± 0.0	0.39 ± 0.0	0.28 ± 0.0	0.48 ± 0.0
Five Cl.	**0.13 ± 0.1**	0.45 ± 0.0	0.05 ± 0.0	0.04 ± 0.0	0.27 ± 0.0	0.41 ± 0.0	0.0 ± 0.0	0.41 ± 0.0
Email	0.1 ± 0.0	0.21 ± 0.0	**0.06 ± 0.0**	0.11 ± 0.0	0.14 ± 0.0	0.2 ± 0.0	0.21 ± 0.0	0.21 ± 0.0

Table 11.14: Average difference in clustering coefficient distribution measured by Earth Mover's Distance between target graphs and random graphs generated by generative graph models. Comparison between global generative graph models and bechmark models that have comparable entropy.

Graph	Walk/0	Unif/10	NG/S/SE	NG/C/SE	SpecG	Config	SBM	Kron
Football	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
NetSci	0.01 ± 0.0	0.01 ± 0.0	0.01 ±0.0	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0
Airport	**0.0 ± 0.0**	0.0 ± 0.0	**0.0 ± 0.0**	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Five Cl.	0.01 ± **0.0**	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	0.01 ± 0.0	**0.0 ± 0.0**	0.01 ± 0.0
Email	**0.0 ± 0.0**	0.0 ± 0.0	**0.0 ± 0.0**	0.0 ± 0.0	**0.0 ± 0.0**	**0.0 ± 0.0**	0.0 ± 0.0	0.0 ± 0.0

Table 11.15: Average difference in betweenness centrality distribution measured by Earth Mover's Distance between target graphs and random graphs generated by generative graph models. Comparison between global generative graph models and bechmark models that have comparable entropy.

Graph	Walk/0	Unif/10	NG/S/SE	NG/C/SE	SpecG	Config	SBM	Kron
Football	1.73 ± 0.1	2.78 ± 0.4	1.66 ± 0.2	1.05 ± 0.1	2.52 ± 0.3	**0.48 ± 0.1**	1.87 ± 0.2	2.44 ± 0.2
NetSci	0.75 ± 0.2	0.41 ± 0.1	0.27 ± 0.1	0.49 ± 0.1	0.88 ± 0.2	**0.08 ± 0.0**	0.96 ± 0.1	0.55 ± 0.1
Airport	3.41 ± 0.2	2.83 ± 0.5	**1.15 ± 0.4**	4.2 ± 1.1	2.52 ± 0.1	2.0 ± 0.1	4.93 ± 0.1	4.69 ± 0.1
Five Cl.	10.67 ± 5.8	4.16 ± 1.6	5.94 ± 0.5	5.47 ± 0.6	13.03 ± 4.1	2.42 ± 0.1	**0.46 ± 0.2**	3.08 ± 0.2
Email	0.7 ± 0.1	1.56 ± 0.3	0.65 ± 0.2	1.84 ± 0.2	0.57 ± 0.2	**0.14 ± 0.0**	2.06 ± 0.1	5.15 ± 0.1

Table 11.16: Average difference in degree distribution measured by Earth Mover's Distance between target graphs and random graphs generated by generative graph models. Comparison between global generative graph models and bechmark models that have comparable entropy.

Graph	Walk/0	Unif/10	NG/S/SE	NG/C/SE	SpecG	Config	SBM	Kron
Football	**115 ± 0**	**115 ± 0**	**115 ± 0**	**115 ± 0**	**115 ± 0**	**115 ± 0**	**115 ± 0**	128 ± 0
NetSci	339 ± 12	365 ± 5	367 ± 9	362 ± 9	330 ± 25	**379 ± 1**	375 ± 2	439 ± 7
Airport	494 ± 2	441 ± 41	476 ± 18	479 ± 11	487 ± 5	499 ± 2	**499 ± 1**	481 ± 4
Five Cl.	**500 ± 0**	**500 ± 0**	428 ± 83	497 ± 7	436 ± 98	**500 ± 0**	323 ± 113	512 ± 0
Email	1113 ± 4	1001 ± 26	1068 ± 26	1117 ± 6	1128 ± 7	1130 ± 2	**1132 ± 1**	893 ± 11

Table 11.17: Size of largest conneced components of random graphs generated by generative graph models. Comparison between global generative graph models and bechmark models that have comparable entropy.

Chapter 12

Implementation Details

In this chapter we provide any code bases used as well as the hyper-paramaters used to generate all plots and tables.

12.1 Spectral Generation

In our implementation of the spectral fitting algorithm (Algorithm 1), to solve LP (7.1) we use $\epsilon = .0001$.

In our implementation of matrix rounding (Section 7.5), we use a constant of $c = .0001$ and a budget z equal to $\frac{1}{4}$ of the number of edges crossing the critical cut.

12.2 Netgan

We train with a batch size measured by the number of *edges* (for our experiments: 2700 edges) rather than the number of *walks* so that in experiments with different walk lengths, the total number of edges seen by the generator is normalized. For each experiment we report, we trained 13 GAN models. During training, every 500 iterations, we drew 15M random walks to construct $\mathcal{W}$ and evaluate the stopping criterion. Once the stopping criterion (Edge Prediction (EP) or Spectrum+Edge Prediction (SP+EP)) was met, we drew 40 graphs each using fixed-edge iterative sampling (FE) and edge-independent sampling (EI). For computing the FMMC, much of our code was adapted from Yang (2015).

We also compare the NetGAN variants to two additional benchmark generative graph models: the Configuration model Bender and Canfield (1978) and Stochastic Block Model Holland et al.

(1983); Karrer and Newman (2011) (SBM).

12.3 Random walk generation

In our implementation of Algorithm 18, we use a minimum size of 5 for number of nodes that must be seen by the seed walks. Within each walk algorithm iteration, we sample a maximum number of 1000 walks. For our implementation of Algorithm 19, we increment entries $\tilde{a}_{u,v}$ for nodes that appear at maximum steps 20 apart.

12.4 Cut fix generation

In our implementation of Algorithm 26, we initialize $GRASP$ with $\beta = .2$ fraction of nodes placed randomly in greedy construction (Algorithm 24). We use a sub-neighborhood of size $\alpha = .5$ fraction of nodes (Algorithm 23).

In our implementation of Algorithm 27, the paramamater for making progress is $\epsilon = .001$. The constant c for relaxing the median is .1.

12.5 Benchmark models

For all figures reported on the benchmark models, we drew 40 random graphs.

12.5.1 Configuration model

The configuratin model is descrined in Section 4.1.1. We use the Networkx implementation of the configuration model Hagberg et al. (2008), omitting all self-loops and multi-edges from the graph.

12.5.2 Stochastic Block model

The Stochastic Block model (SBM) is described in Section 4.2.1. We fit the SBMs with the Sckit learn implementation of spectral clustering Pedregosa et al. (2011), using five clusters for the Five Cluster graph and two clusters for all others.

12.5.3 Kronecker model

The Kronecker model is described in Section 4.2.2. We use the SNAP library for fitting the Kronecker models and generating the Kronecker graphs from these models (Leskovec and Sosič, 2016). For fitting the Kronecker models, we use initiator matrices K of size 2×2 and intialize the matrices at $[[.9, .6], [.6, .1]]$. We use 100 gradient descent iterations. To generate the graphs, we generate graphs of size $n = 2^k$ where k is the celing of $\log(n^*)$ using $K^{[k]}$ as a probabilistic graph.

Chapter 13

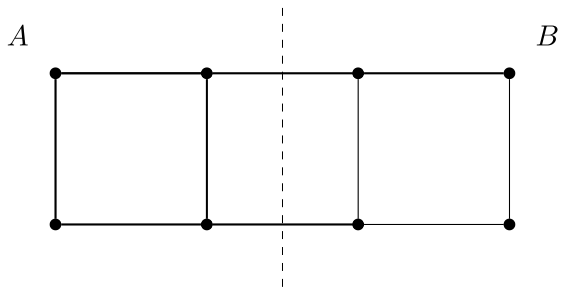
Random graphs using local features and conductance

Chung et.al. show that with high-probability that random graphs subject to any degree distribution are indeed *expanders* which are graphs with high conductance (Chung et al., 2004). Conductance occurs because the number of ways to wire edges such that they are clustered is small compared to the ways to place edges such that they expand (Definition 5.0.3). One way to interpret conductance is the probability that a randomly chosen edge adjacent to a cut will cross. This suggests that if we can bias stub wiring in the configuration model to encourage clustering, then expected conductance should only decrease. One way to encourage clustering is to fix edges into local motifs instead of placing them arbitrarily. This reflects the intuition that edges are more likely to form between nodes if they have common neighbors.

The smallest local motif we can fix is multi-edges. The intuition here is that we fix multi-edges, there are less opportunities for the graph to expand. We look at the probability of being an expander if we fix k multi-edges. For a vertex set V, edge set E' of m' edges are drawn independently from some fixed distribution. We then add k new edges to E' by drawing from the same distribution until we draw k edges that are in E' to add k multi-edges. We show the probability of being an α-expander is monotonically deceasing in k Lemma 12.

Lemma 11. *Let G be drawn subject to a n-dimensional weight vector w such that $m-k$ random edges e are drawn such that $e=(u,v)$ with probability proportional to $w_u w_v$. Add k multi-edges to G by duplicating edges in already drawn in the set of $m-k$. $Pr(\Phi(G) \geq \alpha | w, m, k)$ is monotonically decreasing in k.*

Lemma 12. *Let G_1 and G_2 be drawn according to the model described in lemma 12. $Pr(\Phi(G_1) \geq$*



Add any edge		
Edge Type	Fraction	Conductance
AB	16/28	3/11
AA	6/28	2/12
BB	6/28	2/12
Add triangle edge		
Edge Type	Fraction	Conductance
AB	6/10	3/11
AA	2/10	2/12
BB	2/10	2/12

Figure 13.1: Contrary to our intuition, conditioning on adding triangles does not always increase the probability of producing an expander. Expected conductance when adding an edge to the shown graph is is .227 whereas expected conductance conditioned on adding an edge that forms a triangle is .23.

$\alpha|w, m, k = 1) \geq Pr(\Phi(G_2) \geq \alpha|w, m, k = 0)$

In the spirit of this results, we ideally would like to show (1) fixing larger motifs with high-probability leads to less conductance (2) bound the conductance decrease to understand how many motifs must be fixed to generate graphs that do not expand. However, this appears difficult. We provide an example graph where the expected conductance increases conditioned on adding a triangle (Figure 13).

While we currently lack any general theoretical results relating fixing motifs to conductance, we use Karrer and Newman's generalization of the configuration model to fix motifs up to size 4 and observe what we suspected to be true: fixing motifs up to size 4 is insufficient to prevent generating expanders with high probability (Figure 13.2). We describe Karrer and Newman's model to accommodates arbitrary distributions of sub-graphs in Section 4.1 Karrer and Newman (2010).

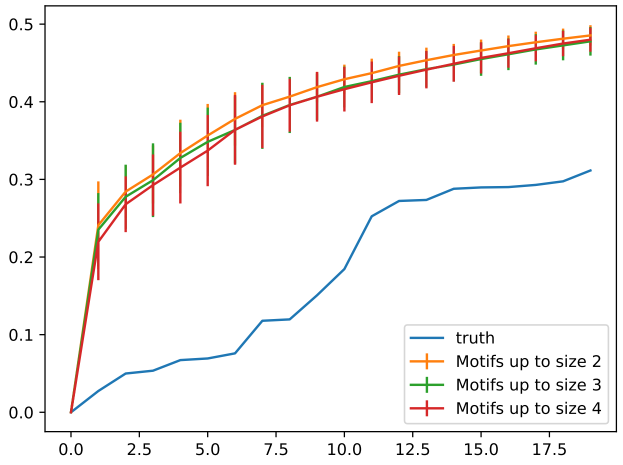


Figure 13.2: Average eigenvalues of the symmetric normalized Laplacian for the Airport graph using the Karrer-Newman configuration model generalization to generate random graphs subject to distributions of motifs. We construct the distribution over motifs by labeling each edge as part of exactly one motif. For each unlabeled edge, we label it with the largest motif it is apart of with other un-labeled edges. We visit the edges adjacent to each node in a random order.

13.1 Proofs for Lemmas

Proof. Proof of Lemma 12 Let $E_1 = E_1' \cup z$ where E_1' are the first m edges drawn and z is the $m+1$-st edge drawn from $V \times V$ according to w. We couple the choices of E_1' and E_2' for G_1 and G_2 so that $E_1' = E_2' = H$ is a set of m edge samples drawn from $(V \times V)$ according to w. We defer the decision of defining H and instead assign value to random-variable $(X_H, \overline{X_H})$ that denotes the minimum-conductance cut of H. Note there could be more than one minimum conductance cuts and we break ties lexicographically. To understand the distribution over minimum-conductance cuts, edge set H takes any value E with probability $\frac{\prod_{i \in H} w_i}{Z^m}$ where $Z = \sum w_i$. Let Y_S denote $E \in \{i^m | i \in (V \times V), (S, \overline{S}) = f(E)\}$ where f computes the minimum-conductance cut of graph (V, E). Then $(X_H, \overline{X_H}) = (S, \overline{S})$ with probability $\frac{|Y_S|}{N^m}$. We now draw random edges z and e conditioning on $(S, \overline{S})$ being the min-cut of H. We order the pairs $V \times V$ such that the $x = |S| \times |\overline{S}|$ take indices $[1, x]$ and the remaining pairs take indices $[x+1, N]$ and we denote the x-th edge as $\pi(x)$. We couple the decision of z and e with random variable j drawn uniformly from $[0, 1]$ so that $z = \pi(x)$ if $j \in (\frac{x-1}{N}, \frac{x}{N}]$ and $e = \pi(x)$ if $j \in (R_S(\pi(x-1)), R_S(\pi(x))]$ where R_S is the CDF of the distribution over $V \times V$ conditioned on $(S, \overline{S})$ being the minimum-conductance cut of H. Suppose

$j \in (R_S(\pi(y-1)), R_S(\pi(y)]$ where $y \leq x$ and $e = \pi(y)$ crosses $(S, \overline{S})$.

Claim 13. *Let $r_S(e)$ denote the probability that $y = e$ conditioned on $(S, \overline{S})$ being a min-conductance cut of (V, H). For all e crossing $(S, \overline{S})$, we have $r_S(e) \leq \frac{1}{N}$ where $N = |V \times V|$ for $m \geq 2$.*

Proof. Let Y_S denote the edge sets that random variable H can take conditioned on $(S, \overline{S})$ being a min-conductance cut. Let $Y_S(e)$ denote the edge sets H can take conditioned on $(S, \overline{S})$ being the min-conductance cut and containing e. Then we can write $r_S(e) = \frac{|Y_S(e)|}{|Y_S|}\frac{1}{m}$ and it suffices to show that $\frac{|Y_S(e)|}{|Y_S|} \leq \frac{m}{N}$. Remember that e crosses $(S, \overline{S})$ so for any $E \in Y_S(e)$, it must be that $(E \setminus e) \cup \{e'\} \in Y_S$. Thus, $\frac{|Y_S(e)|}{|Y_S|} \leq \frac{1}{N-1}$. Thus for $m \geq 2$, we have $\frac{|Y_S(e)|}{|Y_S|} \leq \frac{m}{N}$. □

Claim 14. *Let $r_S(e)$ denote the probability that $y = e$ conditioned on $(S, \overline{S})$ being a min-conductance cut of (V, H). For all e crossing $(S, \overline{S})$, we have $r_S(e) \leq \frac{1}{N}$ where $N = |V \times V|$ for $m \geq 2$.*

Proof. Let Y_S denote the edge sets that random variable H can take conditioned on $(S, \overline{S})$ being a min-conductance cut. Then we can write

$$\begin{aligned} r_S(E) &= \sum_{H \in Y_S s.t. e \in H} Pr(H|S)\frac{w_e}{Z_H} \\ &= \frac{1}{\sum_{H' \in Y_S} Pr(H')} \sum_{H \in Y_S s.t. e \in H} Pr(H)\frac{w_e}{Z_H} \end{aligned}$$

To show $r_S(E) \leq \frac{w_e}{Z}$, it suffices to show

$$\sum_{H \in Y_S s.t. e \in H} Pr(H)\frac{1}{Z_H} \leq \frac{1}{Z}\sum_{H' \in Y_S} Pr(H')$$

We focus on any particular $H \in Y_S$ that contains e. Because e crosses the minimum-conductance cut, $(S, \overline{S})$ is the minimum-conductance cut for $(H \setminus e) \cup \{e'\}$ any edge e'. Let $Y_S(e, H) = \{H \setminus e \cup \{e'\} | H \in Y_S, e \in H\}$. Re-writing the sum in the desired inequality using this fact:

$$\sum_{H \in Y_S s.t. e \in H} \frac{1}{Z_H} \prod_{i \in H} \frac{w_i}{Z} \leq \frac{1}{Z} \Big(\sum_{H' \in \{Y_S(e,H)\}_{H \in Y_S}} \prod_{i \in H'} \frac{w_i}{Z} + \sum_{H' \in Y_S \setminus \{Y_S(e,H)\}_{H \in Y_S}} \prod_{i \in H'} \frac{w_i}{Z} \Big)$$
$$\sum_{H \in Y_S s.t. e \in H} \frac{1}{Z_H} \prod_{i \in H} w_i \leq \frac{1}{Z} \Big(\sum_{H' \in \{Y_S(e,H)\}_{H \in Y_S}} \prod_{i \in H'} w_i + \sum_{H' \in Y_S \setminus \{Y_S(e,H)\}_{H \in Y_S}} \prod_{i \in H'} w_i \Big)$$

We bound every term in the right-hand side by a term on the left-hand side:

$$\frac{1}{Z_H} \prod_{i \in H} w_i \leq \frac{1}{Z} \sum_{H' \in Y_S(e,H)} \prod_{i \in H'} w_i$$
$$\frac{1}{Z_H} w_e \prod_{i \neq e \in H} w_i \leq \frac{1}{Z} \sum_{e'} w_{e'} \prod_{i \neq e \in H} w_i$$
$$\frac{1}{Z_H} w_e \leq 1$$

□

From the claim, we have that $j \leq R_S(y) \leq \frac{y}{N} \leq \frac{x}{N}$, so z crosses $(S, \overline{S})$ if e crosses. Thus, in the event that $\phi(G_2) \geq \alpha$, it must also be that $\phi(G_1) \geq \alpha$ which proves the Lemma. The last step is showing that we can we can construct G_1 and G_2 conditioned on $e \in H$ and $(S, \overline{S})$. Using notation from Claim 2, let $Y_S(e)$ denote the edge sets H can take conditioned on $e \in H$ and $(S, \overline{S})$. To construct G_1 and G_2, H is sampled uniformly from this set. □

Proof. Proof of Lemma 11 The proof is similar to that of Lemma 12. For any two graphs G_1 and G_2, suppose G_1 has q fixed multi-edges and G_2 has $q + k$ fixed multi-edges where $k \geq 1$. It suffices to show that $Pr(\Phi(G_2) \geq \alpha) \geq Pr(\Phi(G_1) \geq \alpha)$. We first couple the choices of the q multi-edges they share conditioning on a min-cut S each time. Next we choose k additional edges to add to G_2 conditioning on the fact that they have already been drawn and k edges to add to G_1. Again, at each step we condition on a min-cut S. From Lemma 1, if the edge we add to G_2 crosses $(S, \overline{S})$ then so does the edge we add to G_1. Finally, we draw the remaining $m - k - q$ edges. □

Chapter 14

Conclusion

This thesis presents three new generative graph models that are built to capture graph connectivity structure using (1) symmetric normalized Laplacian spectra (2) random walks and (3) connectivity across graph cuts. Matching global features is a departure from a long history of generative graph models built around local features or a high-level partition. All three models are built using intricate heuristics that have been extensively tested to understand how best to approach generating graphs subject to global features. In addition to these three new models, we provide new variants of NetGAN by Bojchevski et al. (2018) that place an emphasis on global connectivity structure and can help keep graphs with sparse cuts connected when the classic NetGAN can not. Not only do the methods we provide show that we can generate graphs subject to approximately matching global features, but these methods perform comparably and sometimes much better at matching a number of other graph features of interest, like shortest-path distance, compared to a number of benchmarks. Our generative graph models can compete with the benchmark models not only by matching graph features, but also in the goal of generating a diverse set of graphs measured by the entropy of the edge probabilities.

Generating graphs subject to global features is a relatively new concept in generative graph model research, as such, there are a number of directions for future work. For all of our methods, faster heuristics are of immediate interest. The fastest of the three models is cut fix generation, which scales up to tens of thousands of nodes and requires time on the order of one day. This is a significant improvement over Spectral Generation or NetGAN which (1) Spectral generation can not scale beyond a couple thousand nodes due to the number of eigendecompositions and (2) NetGAN can not make signficant progress within a day once graphs reach size a couple thousand. However, many graphs of interest are on the order of millions of nodes, and for these graphs cut fix generation would not be able to accomodate. Another is exploring simpler heuristics that have theoretical guarantees. We explore further opportunities for future work below.

14.1 Formalizing trade-off between similarity and diversity

We measure how well our graphs match the similarity objective by how well they match graph features of a target graph and diversity based on the entropy of the edge probabilities with which we independently include edges during graph generation. We see experimentally that there seems to be a trade-off between similarity and diversity. For most of our models, the heuristics aim to make progress on the similarity objective by fixing more fixtures. We observe that the more features we fix, the less entropy the model has. Is there a way to formalize this trade-off? Is there a specific graph feature where a model that matches more of these features or matches these fetures closer must come at an entropy cost? Is there another diversity measure that makes this trade-off explicit?

14.2 Formalizing trade-off between matching global and local features

The initial motivation for our work raises a very intriguing general direction. We know generating graphs subject to most degree distributions produces a graph with high conductance (Bollobás, 1998). We perform experiments that suggest that even when local properties other than the degree distributions (such as triangle counts and other small motifs) of a graph are fixed, generating graphs under such constraints still produces expander graphs with high probability (Chapter 13). Without any qualifications, this claim is definitely false. For example, a 3-regular graph in which each node participates in 3 triangles must be the union of disjoint 4-cliques, not an expander. However, under a careful formalization, we believe that such a claim should hold true, and it would be desirable to corroborate the intuition that local motifs do not constrain global structure enough to prevent most graphs from being expanders.

Bibliography

P.-A. Absil and Jérôme Malick. Projection-like retractions on matrix manifolds. *SIAM Journal on Optimization*, 22(1):135–158, 2012.

P.-A. Absil, Robert Mahony, and Rodolphe Sepulchre. *Optimization algorithms on matrix manifolds.* Princeton University Press, 2009.

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

Martín Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *Intl. Conf. on Machine Learning*, pages 214–223, 2017.

Lars Backstrom and Jure Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 635–644, 2011.

Luca Baldesi, Carter T Butts, and Athina Markopoulou. Spectral graph forge: Graph generation targeting modularity. pages 1727–1735. IEEE, 2018.

Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286 (5439):509–512, 1999.

Edward A. Bender and E. Rodney Canfield. The asymptotic number of labelled graphs with given degree sequences. *Journal of Combinatorial Theory (A)*, 24:296–307, 1978.

Yoshua Bengio. Artificial neural networks and their application to sequence recognition. 1993.

Noam Berger, Christian Borgs, Jennifer T. Chayes, and Amin Saberi. On the spread of viruses on the internet. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 301–310. SIAM, 2005. URL `http://dl.acm.org/citation.cfm?id=1070432.1070475`.

Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. NetGAN: Generating graphs via random walks. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, volume 80, pages 610–619, 2018.

Béla Bollobás. Random graphs. In *Modern Graph Theory*, pages 215–252. Springer, 1998.

Béla Bollobás and Oliver Riordan. Robustness and vulnerability of scale-free random graphs. *Internet Mathematics*, 1(1):1–35, 2004.

Christian Borgs, Jennifer T Chayes, László Lovász, Vera T Sós, and Katalin Vesztergombi. Convergent sequences of dense graphs i: Subgraph frequencies, metric properties and testing. *Advances in Mathematics*, 219(6):1801–1851, 2008.

Alberto Borobia and Roberto Canogar. The real nonnegative inverse eigenvalue problem is np-hard. *Linear Algebra and its Applications*, 522:127 – 139, 2017.

Stephen Boyd, Persi Diaconis, and Lin Xiao. Fastest mixing markov chain on a graph. *SIAM review*, 46(4):667–689, 2004.

Tom Britton, Svante Janson, and Anders Martin-Löf. Graphs with specified degree distributions, simple epidemics, and local vaccination strategies. *Advances in Applied Probability*, 39(4):922–948, 2007.

Steve Butler and Jason Grout. A construction of cospectral graphs for the normalized laplacian. *The Electronic Journal of Combinatorics*, 18(1):231, 2011.

Chen Chen, Hanghang Tong, B. Aditya Prakash, Tina Eliassi-Rad, Michalis Faloutsos, and Christos Faloutsos. Eigen-optimization on large graphs by edge manipulation. *Intl. Conf. on Knowledge Discovery and Data Mining*, 10(4):49, 2016.

Moody T. Chu and Gene H. Golub. *Inverse Eigenvalue Problems: Theory, Algorithms, and Applications*. Oxford University Press, 2005.

Fan Chung and Mary Radcliffe. On the spectra of general random graphs. *the electronic journal of combinatorics*, 18(1):215, 2011.

Fan Chung, Linyuan Lu, and Van Vu. The spectra of random graphs with given expected degrees. *Internet Mathematics*, 1(3):257–275, 2004.

Fan R. K. Chung and Linyuan Lu. The average distance in random graphs with given expected degrees. *Proc. Natl. Acad. of Science*, 99:15879–15882, 2002.

Fan R. K. Chung, Linyuan Lu, and Van Vu. The spectra of random graphs with given expected degrees. 100(11):6313–6318, 2003.

Fan RK Chung and Fan Chung Graham. *Spectral graph theory*. Number 92. American Mathematical Soc., 1997.

Aaron Clauset, Ellen Tucker, and Matthias Sainz. *The Colorado Index of Complex Networks*, 2016. URL https://icon.colorado.edu/.

Nicola De Cao and Thomas Kipf. MolGAN: An implicit generative model for small molecular graphs. *ICML 2018 Workshop on Theoretical Foundations and Applications of Deep Generative Models*, 2018.

Carl Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

Stanley C. Eisenstat and Ilse C. F. Ipsen. Relative perturbation techniques for singular value problems. *SIAM Journal on Numerical Analysis*, 32(6):1972–1988, 1995.

Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, 5:17–61, 1960.

Thomas A Feo and Mauricio GC Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

Matthew Fickus, Dustin G. Mixon, Miriam J. Poteet, and Nate Strawn. Constructing all self-adjoint matrices with prescribed spectrum and diagonal. *Advances in Computational Mathematics*, 39 (3-4):585–609, 2013.

Miroslav Fiedler. Eigenvalues of nonnegative symmetric matrices. *Linear Algebra and its Applications*, 9:119–142, 1974.

Francois Fouss, Alain Pirotte, Jean-Michel Renders, and Marco Saerens. A novel way of computing dissimilarities between nodes of a graph, with application to collaborative filtering and subspace projection of the graph nodes. 2006.

Ove Frank and David Strauss. Markov graphs. *Journal of the American Statistical Association*, 81 (395):832–842, 1986.

Carolina Fransson and Pieter Trapman. Sir epidemics and vaccination on random graphs with clustering. *Journal of mathematical biology*, 78(7):2369–2398, 2019.

Alan Frieze and Ravi Kannan. Quick approximation to matrices and applications. *Combinatorica*, 19(2):175–220, 1999.

Rajiv Gandhi, Samir Khuller, Srinivasan Parthasarathy, and Aravind Srinivasan. Dependent rounding and its applications to approximation algorithms. *Journal of the ACM*, 53(3):324–360, 2006.

Ayalvadi Ganesh, Laurent Massoulié, and Don Towsley. The effect of network topology on the spread of epidemics. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 2, pages 1455–1466. IEEE, 2005.

George Giakkoupis. Tight bounds for rumor spreading in graphs of a given conductance. In Thomas Schwentick and Christoph Dürr, editors, *28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011, March 10-12, 2011, Dortmund, Germany*, volume 9 of *LIPIcs*, pages 57–68. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. doi: 10.4230/LIPIcs. STACS.2011.57. URL https://doi.org/10.4230/LIPIcs.STACS.2011.57.

Minas Gjoka, Maciej Kurant, and Athina Markopoulou. 2.5 k-graphs: from sampling to generation. pages 1968–1976. IEEE, 2013.

Chris D. Godsil and Brendan D. McKay. Constructing cospectral graphs. *Aequationes Mathematicae*, 25(1):257–268, 1982.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In *Advances in neural information processing systems*, pages 5767–5777, 2017.

Alexander Gutfraind, Ilya Safro, and Lauren Ancel Meyers. Multiscale network generation. In *2015 18th international conference on information fusion (fusion)*, pages 158–165. IEEE, 2015.

Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Paul W. Holland, Kathryn B. Laskey, and Samuel Leinhardt. Stochastic blockmodels: Some first steps. *Social Networks*, 5:109–137, 1983.

Roger A. Horn and Charles R. Johnson. *Matrix Analysis.* Cambridge University Press, 1990.

Matthew O Jackson. *Social and economic networks.* Princeton university press, 2010.

Mark Jerrum and Alistair Sinclair. Approximating the permanent. *SIAM journal on computing*, 18 (6):1149–1178, 1989.

Brian Karrer and Mark EJ Newman. Random graphs containing arbitrary distributions of subgraphs. *Physical Review E*, 82(6):066118, 2010.

Brian Karrer and Mark EJ Newman. Stochastic blockmodels and community structure in networks. *Physical review E*, 83(1):016107, 2011.

Tosio Kato. Variation of discrete spectra. *Communications in Mathematical Physics*, 111:501–504, 1987.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Durk P Kingma, Shakir Mohamed, Danilo Jimenez Rezende, and Max Welling. Semi-supervised learning with deep generative models. In *Advances in neural information processing systems*, pages 3581–3589, 2014.

Vikram Krishnamurthy and Buddhika Nettasinghe. Information diffusion in social networks: friendship paradox based models and statistical inference. In *Modeling, Stochastic Control, Optimization, and Applications*, pages 369–406. Springer, 2019.

Solomon Kullback. *Information theory and statistics*. Courier Corporation, 1997.

Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tomkins, and Eli Upfal. Stochastic models for the web graph. pages 57–65. IEEE, 2000.

Thomas J. Laffey and Helena Šmigoc. Construction of nonnegative symmetric matrices with given spectrum. *Linear algebra and its applications*, 421(1):97–109, 2007.

James R Lee, Shayan Oveis Gharan, and Luca Trevisan. Multiway spectral partitioning and higher-order cheeger inequalities. *Journal of the ACM (JACM)*, 61(6):37, 2014.

Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology*, 8(1):1, 2016.

Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. 1(1):2, 2007.

Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11 (Feb):985–1042, 2010.

David A. Levin and Yuval Peres. *Markov Chains and Mixing Times.* American Mathematical Society, 2017.

Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018.

David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.

Dunia López-Pintado et al. Diffusion in complex social networks.

Russell Merris. Large families of laplacian isospectral graphs. *Linear and Multilinear Algebra*, 43 (1–3):201–205, 1997.

Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 29–42, 2007.

Michael Molloy and Bruce Reed. A critical point for random graphs with a given degree sequence. *Random structures & algorithms*, 6(2-3):161–180, 1995.

Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1990.

Mark E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. 74, 2006.

Mark E. J. Newman. Random graphs with clustering. *Physical Review Letters*, 103(5):058701, 2009.

Mark E. J. Newman, Steven H. Strogatz, and Duncan J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E*, 64(2):026118, 2001.

Jorge Nocedal and Stephen Wright. *Numerical Optimization.* Springer, 2006.

Chiara Orsini, Marija M. Dankulov, Pol Colomer-de Simón, Almerima Jamakovic, Priya Mahadevan, Amin Vahdat, Kevin E. Bassler, Zoltán Toroczkai, Marián Boguñá, Guido Caldarelli, Santo Fortunato, and Dmitri Kiroukov. Quantifying randomness in real networks. *Nature Communications*, 6:8627, 2015.

Michael L Overton and Robert S Womersley. Optimality conditions and duality theory for minimizing sums of the largest eigenvalues of symmetric matrices. *Mathematical Programming*, 62(1-3):321–357, 1993.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

David Martin Powers. Evaluation: From precision, recall and f-measure to roc, informedness, markedness and correlation. *Intl. J. of Machine Learning Technology*, 2(1):37–63, 2011.

Stephen Ranshous, Shitian Shen, Danai Koutra, Steve Harenberg, Christos Faloutsos, and Nagiza F Samatova. Anomaly detection in dynamic networks: a survey. *Wiley Interdisciplinary Reviews: Computational Statistics*, 7(3):223–247, 2015.

Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40(2):99–121, 2000.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55, 2001.

Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

Alana Shine and David Kempe. Generative graph models based on laplacian spectra? In *The World Wide Web Conference*, WWW '19, page 1691–1701, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366748. doi: 10.1145/3308558.3313631. URL https://doi.org/10.1145/3308558.3313631.

Martin Simonovsky and Nikos Komodakis. Graphvae: Towards generation of small graphs using variational autoencoders. In *International Conference on Artificial Neural Networks*, pages 412–422. Springer, 2018.

Daniel A Spielman and Shang-Hua Teng. A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning. *SIAM Journal on computing*, 42(1):1–26, 2013.

Christian L Staudt, Michael Hamann, Alexander Gutfraind, Ilya Safro, and Henning Meyerhenke. Generating realistic scaled complex networks. *Applied network science*, 2(1):36, 2017.

Cédric Villani. *Optimal transport: old and new*, volume 338. Springer Science & Business Media, 2008.

Duncan J. Watts and Steven Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393: 440–442, 1998.

Zaiwen Wen and Wotao Yin. A feasible method for optimization with orthogonality constraints. *Mathematical Programming*, 142(1–2):397–434, 2013.

David White and Richard C. Wilson. Spectral generative models for graphs. In *Proc. Intl. Conf. on Image Analysis and Processing*, pages 35–42, 2007.

David H. White. *Generative Models for Graphs.* PhD thesis, 2009.

Richard C Wilson and Ping Zhu. A study of graph spectra for comparing graphs and trees. *Pattern Recognition*, 41(9):2833–2841, 2008.

Bai Xiao and Edwin R. Hancock. A spectral generative model for graph structure. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 173–181, 2006.

Yang Kjeldsen Yang. Numerical methods for solving the fastest mixing markov chain problem. Master's thesis, University of Oslo, 2015.

Zhijun Yin, Manish Gupta, Tim Weninger, and Jiawei Han. A unified framework for link recommendation using random walks. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 152–159. IEEE, 2010.

Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, volume 80, pages 5708–5717, 2018.