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

Table of Contents

Acknowledgements

I would like to thank my advisor David Kempe for his guidance throughout my Ph.D. I would also like to thank my committee, Aram Galstyan, Xiang Ren, and Kayla de la Haye, for their advice during the preparation of this dissertation.

List of Tables

List of Figures

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 transmission (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.

Topologies of graphs are critical in determining how random processes, like disease spread, behave. In particular, graph *connectivity* is especially important for determining how processes like disease spread and information flow behave (Berger et al., 2005; Ganesh et al., 2005). One connectivity measure related to these processes is called *conductance*. 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 the connectivity across a bottleneck cut determines how easy 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) 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 instead of 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 uniformly from the set of all graphs subject to the given 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. Moreover, what the researcher observes in the simulations may contradict what happens in the real world. 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 node degrees or number of triangles, or a high-level community structure. This thesis is a departure from this line of study and describes ways to generates graphs subject to matching global features capable of capturing complex connectivity structure. This thesis presents three new ways to generate graphs using three different global features:

1. Symmetric normalized Laplacian spectra.

2. Random walks.

3. Cut connectivity.

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 so that they are useful in down-stream tasks. We find that our generative graph models are competitive and often perform better in the similarity 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 prevalence of 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. Generative graph models 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 (Jackson, 2010), biological (Pavlopoulos et al., 2011; Guo and Zhao, 2020) and computer networks (Erciyes, 2013; Boukerche, 2008). How these datasets can be described as nodes and edges is in Table 1.1.

Real-world data	Node	Edges
Social network	People	Communication or interaction
Biological network	Cells, proteins or genes	Interaction
Computer network	Computers	Cable or communication

Table 1.1: Real-world datasets as graphs.

1.1.1 Generative graph models for social networks

One application of generative graph models for social networks is running simulations of *contagion* processes (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 are studied 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 (or another process simulated on graphs). 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 use the high-probability statements about the random graphs generated 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).

1.1.2 Generative graph models for biological networks

Guo and Zhao (2020) survey a number of applications for generative graph models for biological networks; we explore some of them here. Molecule generation is an important problem in drug discovery and can be computationally challenging due to its combinatorial nature. One solution is to model the molecules as a graph with nodes/edges representing atoms/bonds and training a generative graph model to generate graphs that resemble molecules (De Cao and Kipf, 2018). One type of molecule that biologists study are proteins. Protein problems include protein structure prediction and protein design. Anand and Huang (2018) design a generative graph model specifically for protein graphs.

1.1.3 Generative graph models for computer networks

One application for generative graph models for computer networks is understanding how robust networks are to attacks modeled as node deletions. We can study how often random graphs drawn from various models are disconnected by node deletion to help explain what properties are responsible for making a network robust to an attack (Bollobás and Riordan, 2004). The robust models can then be used to design robust computer networks (Buesser et al., 2011). Another application is to understand how fixing one graph property affects another; this can be useful in detecting graph anomalies in the presence of adversarial behavior (Ranshous et al., 2015). Generative graph models often generate graphs subject to fixing graph properties, and by generating many graphs from these models we can start to understand how fixing one affects the others.

1.2 Overview 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 or more 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 to be similar to a set of real-world graphs, often using 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 the 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. One alternative is to use a data set of "similar" graphs and generate graphs that resemble these. With multiple examples, learning techniques can be used to extract features common to graphs in the graph data set and generate new graphs that share these features. One tool to extract these common features is deep neural networks (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

While many generative graph models explicitly or implicitly aim to match *local* graph features, there are significantly fewer generative graph models explicitly built to match *global* features that capture the connectivity structure of a graph. This move toward global features is motivated by the fact that random graphs subject to most degree distributions with high probability are highly connected (Bollobás, 1998; Chung et al., 2004). Thus to study real-world graphs that lack highly connected structure and generate graphs that are sparsely connected, we need alternative models. Beyond degrees, we explore the effect of fixing larger local features like motifs in Chapter 13. Experiments suggest this does little to preserve sparsely connected structure.

Perhaps the most basic notion of connectivity is defined using cuts. The connectivity of a cut is the number of edges between two sides of a cut. A distribution over the connectivity across cuts would describe the density of sparse cuts to dense ones. 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. 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. By measuring the smallest ratio, conductance captures the connectivity of the "bottleneck" cut. It gives us a lower bound on the connectivity across all cuts in the graph. This thesis explores generative graph models that sample from sets of graphs with similar conductance to sample from sets of graphs with similar connectivity.

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.

Spectrum-matching 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 the Laplacian (Theorem 5.1.3) and more intricate results show that higher-order eigenvalues characterize the connectivity across additional sparse cuts (Lee et al., 2014). The goal of Spectrum-matching 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-based generation* inspired by NetGAN proposed by Bojchevski et al. (2018). A *walk* on a graph is a sequence of nodes where each consecutive node pair is an edge in the graph. A *random walk* is a random process that generates a walk where each node that appears is drawn from a distribution depending on the previous nodes appearing on the walk. 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 frequencies with which 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 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-based 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 a 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-based generation by defining cuts based on how many times nodes appear together on random walks.

Our last generative graph model matches the connectivity across cuts directly (Chapter 10). *Cut fix generation* corrects cuts in some "seed" graph to match the connectivity of those cuts in the input graph. The seed 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 seed 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. This lessens the number of cuts that need to be corrected in order for the corrected seed 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 lengths and clustering coefficients. 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 perfectly by memorizing the

input graph and outputting the input graph every time. However, this type of model can not be used to make general statements about any class of graphs the input graph belongs to. The *diversity* objective aims to generate outputs from a large set of graphs. Moreover, several graphs 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. Many of our graph models 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 generated graphs. We summarize the graph features matched to achieve similarity and the variation tools used to achieve diversity in Table 1.2.

Graph Generator	Graph Feature to Match	Variation Tool
Spectrum-matching generation	Laplacian Spectra	Random Basis
NetGAN (and variants)	Random Walks	Noisy walk distribution (Learned)
Random walk-based generation	Random Walks	Noisy walk distribution (Edges inserted)
Cut fix generation	Cut Connectivity	Subset of cuts matched

Table 1.2: Each generative graph model achieves similarity by matching a graph feature, and diversity by applying a variation tool.

Chapter 2

Notation and central concepts

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.

$\boldsymbol{e}_i^n$ is the i-th n-dimensional basis 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 $d_{v,v} = \sum_{u \in \mathcal{V}} x_{u,v}$.

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 $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. 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 2.2.1).

Definition 2.2.1 (Fiedler cut)**.** For the 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 $x_{v^*} = \arg\min_{v \in \mathcal{V}} \varphi_{\mathcal{G}}(Q(\mathbf{x}, x_v))$ and $\varphi_{\mathcal{G}}(S)$ denotes the conductance of $S \subseteq \mathcal{V}$ (Definition 5.1.2).

A cut of a graph with vertices $\mathcal{V}$ is a partition of $\mathcal{V}$ denoted by $(\mathcal{S}, \overline{\mathcal{S}})$ with $\mathcal{S} \subseteq \mathcal{V}$ and $\overline{\mathcal{S}} = \{v \in \mathcal{V} : v \notin \mathcal{S}\}$.

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 *probabilistic adjacency matrix* A with entries $a_{v,u} \in [0, 1]$. The matrix A is called a probabilistic adjacency matrix because each entry can be treated as an edge probability for *independent edge sampling* (Definition 2.3.1).

Definition 2.3.1 (Independent edge sampling (Symmetric))**.** Independent edge sampling takes a symmetric matrix A with entries $a_{v,u} \in [0, 1]$ as input 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 $a_{v,u}$.

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

Often, the template matrices $\tilde{A}$ are constructed using using counts of pairs (v, u) stored in $\tilde{a}_{v,u}$. We refer to these non-negative symmetric integer templates as *frequency matrices* and denote them using $\tilde{A}$ as well.

2.4 Random walk notation

A *Markov chain* is a random discrete process (Levin and Peres, 2017). 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 $t > k$, $Pr(\boldsymbol{w}[t] = v|\boldsymbol{w}[t-1], \boldsymbol{w}[t-2], \ldots, \boldsymbol{w}[t-k], \ldots, \boldsymbol{w}[2], \boldsymbol{w}[1]) = Pr(\boldsymbol{w}[t] = v|\boldsymbol{w}[t-1], \boldsymbol{w}[t-2], \ldots, \boldsymbol{w}[t-k])$ (Raftery, 1985).

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 $v \in \mathcal{V}$ (Levin and Peres, 2017). For all $v \in \mathcal{V}$, $p(\boldsymbol{w}[t] = v|\boldsymbol{w}[t-1] = u)$ is non-zero if and only if $(v, u) \in \mathcal{E}$.

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 $t-1$: $p(\boldsymbol{w}[t] = v|\boldsymbol{w}[t-1]) = p(\boldsymbol{w}[t] = v|\boldsymbol{w}[t-1], \boldsymbol{w}[t-2], \ldots, \boldsymbol{w}[2], \boldsymbol{w}[1])$ (Levin and Peres, 2017). 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 $Pr(\boldsymbol{w}[t] = v|\boldsymbol{w}[t-1] = u)$ as a *transition matrix* R so that $r_{u,v} = Pr(\boldsymbol{w}[t] = v|\boldsymbol{w}[t-1] = u)$ and the entries of each row all sum to 1 (Levin and Peres, 2017).

A *stationary distribution* over values a Markov state can take is a vector $\boldsymbol{\pi}$ such that $\boldsymbol{\pi}R = \boldsymbol{\pi}$ with $\pi_v = \lim_{t\to\infty} R'\mathbf{e}_u$ for any $u \in \mathcal{V}$ (Levin and Peres, 2017). Because $\boldsymbol{\pi}$ is a distribution, $\pi_v \geq 0$ for all $v \in \mathcal{V}$ and $\sum_{v\in\mathcal{V}} \pi_v = 1$. A Markov chain is irreducible if the probability that $\boldsymbol{w}[1] = v$ and $\boldsymbol{w}[t] = u$ is positive for some finite t for all pairs of states $(v, u) \in \mathcal{X} \times \mathcal{X}$. The *period* of a state v is the greatest common divisor of the set $\{t : r^t_{v,v} > 0\}$. A Markov chain is aperiodic if the period of all states is 1. If a Markov chain is irreducible and aperiodic, then a unique stationary distribution exists (Levin and Peres, 2017).

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 over support $\mathcal{X}$: *Earth mover's distance* $d_{\mathrm{EMD}}(p, q)$ and *Total variation distance* $d_{\mathrm{TV}}(p, q)$.

The Earth mover's distance between p and q computes a *transport plan* f from p to q with respect to a metric d over the support $\mathcal{X}$ of p and q (Rubner et al., 2000). The transport plan $f_{i,j}$ 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.
2. $q(j) = \sum_{i \in \mathcal{X}} f_{i,j}$ for all j.

The first constraint is to ensure all mass $p(i)$ is transported out of all i and the second is to ensure all mass $q(j)$ is transported into all j. A transport plan f is *feasible* if it meets both constraints. We write the set of all feasible transport plans as $\mathcal{F}$. 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 then $\min_{f \in \mathcal{F}} c(f, d)$.

When $\mathcal{X}$ is one-dimensional Euclidean space, we can compute it using the following algorithm. The minimum cost transport plan will move earth from values in $\mathcal{X}$ close to each other in absolute difference; without loss of generality it is assumed that $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ with $x_i < x_{i+1}$:

- $f_{1,0} = 0$, initial earth moved is zero.
- $f_{i+1,i} = p(i) + f_{i,i-1} - q(i)$, for $i = 1, 2, \ldots, n-1$, compute earth moved from i to $i+1$ where a negative value indicates that 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.

- $d_{\mathrm{EMD}}(p,q) = \sum_{i=1}^{n-1} |f_{i+1,i}|(x_{i+1} - x_i)$, weight the absolute value of the earth moved by the distance.

The algorithm 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 has 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 between p and q is defined as $d_{\mathrm{TV}}(p,q) = \frac{1}{2}\sum_{i\in\mathcal{X}} |p(i) - q(i)| = \max_{\mathcal{A}\subset\mathcal{X}} |\sum_{i\in\mathcal{A}} p(i) - q(i)|$.

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 has dimension $n \times n$.

Chapter 3

Evaluating generative graph models: Similarity and Diversity

A generative graph model samples random graphs $\mathcal{G}$ from the set $\mathcal{H}$ of graphs 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 model 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 path lengths 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 to 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 p places negligible probability on graphs 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 $\mathcal{H}$ only differ from $\mathcal{G}^*$ by one

edge, 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 fewer satisfying graphs. For the definitions of similarity and diversity that we use and our algorithms, we are not aware of a formal trade-off. However, we see experimentally that models that score higher on our similarity objectives come at a diversity cost. For experimental results that show this trade-off, see Sections 8.7.2, 9.7, and 10.4 and Chapter 11.

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 form a distribution. We use the *Earth mover's distance* (EMD) between the two distributions (Section 2.5). 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 efficiently using an algorithm 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 to compute the ℓ_2 norm of $\boldsymbol{\lambda} - \boldsymbol{\lambda}'$. One problem with ℓ_2 is that it weights the differences between all eigenvalues equally. We know from Cheeger's inequality (Theorem 5.1.3) 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). Therefore, to compare spectra we weight each squared difference inversely proportional to its index using the ℓ_2^{LW} semimetric 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 these features are commonly used to compare the similarity of graphs (Bojchevski et al., 2018; Leskovec et al., 2010).

Features	Abbreviation	Description	Metric
Spectrum	Spec	Eigenvalues of the symmetric normalized Laplacian matrix.	ℓ_2^{LW}
Shortest Path Length Distribution	SP	Distribution of lengths of shortest paths from v to u for all vertex pairs (v, u).	EMD
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.	EMD
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 $\sum_{v,u \neq s} \frac{\sigma_{v,u}(s)}{\sigma_{v,u}}$.	EMD
Degree Distribution	Deg	Frequency for each degree. The degree of a node is the number of edges incident on it.	EMD

Table 3.1: Graph features used to compute graph similarity.

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 fairly easily 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 probabilistic adjacency matrix A with entries $a_{u,v} \in [0, 1]$ using independent edge sampling (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 the size of $\mathcal{V}$ 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). This does not compute the entropy of p exactly, but provides a reasonable measure for the diversity of the graphs drawn from p.

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 producing diverse graphs.

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

Chapter 4

Review of generative graph models

Generative graph models are built to generate graphs that resemble real-world graphs. 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.

1. **Prescribed.** 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, features shared could be a degree distribution or a high-level clustering. We call these features *prescribed* because the type of feature is specified.

2. **Dynamic process.** 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.

3. **Data driven.** Alternatively, the features could be learned by extracting 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 a variation mechanism to generate graphs with those features that do not all look the same.[1] For the dynamic process models, the dynamic process itself is often random which provides variation among the graphs. For the feature based models, one method is to randomize the graph features that are not prescribed (prescriptive) or not common to all the graphs in the

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

data set (data driven). 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 observed real-world features slightly, such as edge/node deletion/addition. This is done in many of the high-level clustering schemes. Another perturbation method in 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 in Table 4.1 according to how they identify features to match and the type of features.

Prescriptive (local features)	Config, ERGM (Section 4.1)
Prescriptive (high level partitioning)	SBM, Kronecker (Section 4.2)
Prescriptive (global features)	**Spectrum-matching generation**, **Random walk-based generation**, **Cut fix generation**, ModulGen (Section 4.3)
Dynamic Process	BA, Small-world (Section 4.4)
Data driven (features learned)	Graph VAE, Graph Net, GraphRNN (Section 4.5.2)
Data driven/Prescriptive	NetGAN, MolGAN (Section 4.5.3)

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

There are a number of factors to keep in mind when deciding what type of generative graph model is the most appropriate for a given task. We explain some of these factors and how they relate to the existing models described in this chapter and the new models presented in this thesis in Section 5.7.

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 (Bender and Canfield, 1978; Molloy and Reed, 1995). More generally, configuration models match the number of motifs adjacent to each node. The randomness is because the graph is explicitly drawn (uniformly) randomly from the class of all graphs (or multi-graphs) that have these motif distributions.

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 an 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 on vertex e on the right are 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 appear for cycles of length 4. The degree distribution matches 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 the model of 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 $((v, u), (u, s))$ for which $(v, s) \in E$ and thus forms a triangle. If triangles adjacent to v do not share any edges, $d_v = s_v + 2w_v$ 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 a 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 as the motifs grow larger. Thus, this technique is only used for small motifs and thus local features.

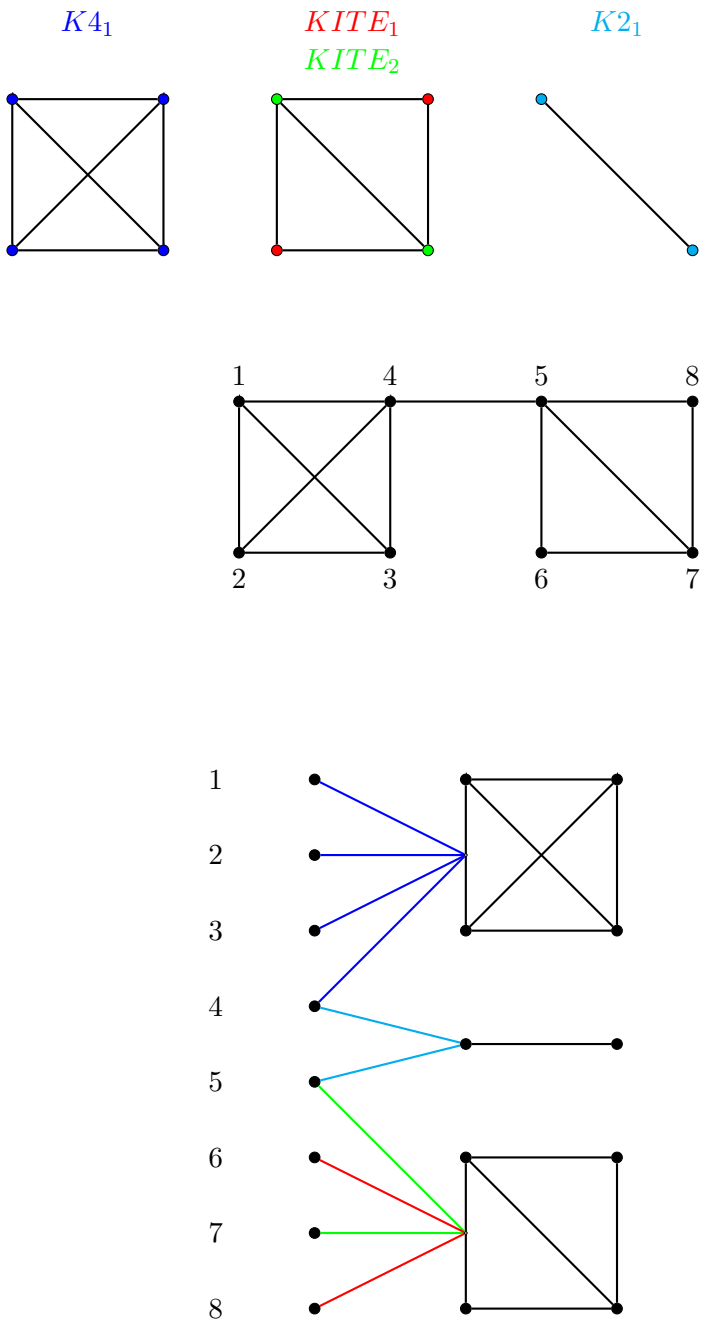


Figure 4.1: Example of configuration model fixing four types of motifs. The first row is each motif and each topological role in that motif. The next row is a graph corresponding to the subsequent matching of topological roles to motifs. Each vertex on the left represents a topological role and is matched 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.

4.1.2 Exponential random graph model

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

4.2 High-level clustering models

We use two high-level clustering models as benchmarks in our experiments (Chapter 11):

1. The *Stochastic block model* (SBM) (Holland et al., 1983; Karrer and Newman, 2011).

2. The *Kronecker* model (Leskovec et al., 2010).

For both, independent edge probabilities are assigned to each node pair (v, u) to build a probababilistic adjacency matrix A and graphs are generated using independent edge sampling (Definition 2.3.1). The most basic generative graph model that generates graph using a probababilistic adjacency matrix A and independent edge sampling is the *Erdős Rényi* random graph model which assigns every pair a uniform edge probability p (Erdős and Rényi, 1960). All pairs are treated the same. High-level clustering models differ from Erdős Rényi because they assign a clustering membership to each node and assign edge probabilities between nodes v and u with respect to their clustering memberships.

The SBM model assigns one of k clusters to each node. All node pairs v and u that belong to the same clusters i and j are assigned the same edge probability $p_{i,j}$. The Kronecker model can be treated as a generalization because nodes are assigned a profile of clustering memberships instead of a single cluster. All nodes in the Kronecker model have different clustering membership profiles, introducing more variation in the edge probabilities.

4.2.1 Stochastic Block Models

The *Stochastic block model* (SBM) and its variants bias toward 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 (v, u) is labeled with an inter-/intra-cluster edge identity corresponding to the clusters that v and u belong to. Graphs are sampled by including an edge for pair (v, u) with the probability prescribed by the inter-/intra-cluster edge identity of (v, u) 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 the correct number of 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 (MacQueen, 1967) 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 v and node u is encoded in $K^{[k]} = K \bigotimes K \cdots \bigotimes K$ 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). The product $K \bigotimes K'$ for 2×2 dimensional K is written as

$$K \bigotimes K' = \begin{bmatrix} K_{11}K' & K_{12}K' \\ K_{21}K' & K_{22}K' \end{bmatrix}$$

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

The initiator matrix K fully parameterizes the Kronecker model. Graphs of size 2^k are generated by using $K^{[k]}$ as a probabilistic adjacency matrix and including each edge (v, u) with probability equal to the matrix entry $k_{v,u}^{[k]}$ (independent edge sampling, Definition 2.3.1). The Kronecker models are fit by maximizing the likelihood of generating $\mathcal{G}^*$ from $K^{[k]}$ over initiator matrices K. Leskovec et al. (2010) maximize the likelihood 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 than 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 affected by those made in the beginning, but as long as self-loops are allowed, whatever choices are made will not prevent matching the remaining local properties. With global features, this seems a lot harder because 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 later on in the matching process.

While there is not as large of a literature on 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, Spectrum-matching 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 7.2.

Matrices other than the symmetric normalized Laplacian can be used to construct graphs as well. Baldesi et al. (2018) propose *ModulGen* that generates 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. 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 indicates that 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 $p_{v,u}$. Graphs are then generated using independent edge sampling (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 generative graph models

The focus of this thesis is on matching global graph features specifically. To better understand the advantages and disadvantages of focusing on global features, we focus our comparisons on other generative graph models that aim to match graph features as well. Dynamic models generally aim to make graphs realistic through dynamics that resemble those in the real world instead of identifying real-world features. Therefore, we did not run experiments using dynamic approaches. We do include a brief discussion of dynamic models in this section.

The preferential attachment model is one of the dynamic models designed to generate graphs with a specific feature (Barabási and Albert, 1999). The preferential attachment model is designed to produce scale-free graphs (i.e., graphs with degree distributions that have a long tail). The idea is that the graph grows by choosing a node proportional to its 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 Data driven generative graph models

Unlike the models discussed in Sections 4.1, 4.2, 4.3, and 4.4 that build a distribution p over graphs from a single graph $\mathcal{G}^*$, the data driven 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 $\mathbf{z}$ and learn the parameters of that process. Data driven generative models circumvent the need to prescribe features that $\mathcal{G}$ should have by building from a data set $\mathcal{H}^*$ instead of a single graph $\mathcal{G}^*$. The models extract common features among the graphs in $\mathcal{H}^*$ and then use these features to build a distribution over $\mathcal{G}$. Often, these models are built using a neural network and these are called *deep generative graph models*.

There are exceptions to data driven 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. NetGAN is also data driven because it uses a data set of graph features from a single graph instead of a data set of graphs. Additionally, NetGAN is prescriptive because the type of graph features in the data set it trains from is specified.

Deep generative graph models have been developed after the success of 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 is given and the task is to generate data that resembles the data set. Deep generative graph models extend the results of 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, they can quickly become intractable as the size of the graph grows because of the size of the matrix. Representing a graph as an adjacency matrix in a neural network can lead to long training times because of the number of weights needed for each entry in the adjacency matrix. Furthermore, whatever graph representation is used 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 to guide the training process. 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 graph structure is what is important which 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

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

A Recurrent Neural Network (RNN) is a chain of feed-forward neural networks (a network without cycles) 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 in the previous step. At time t, $f_{\boldsymbol{\theta}}(X_t, h_{t-1})$ produces a new output state o_t and a new hidden state h_t. 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 that uses special rules for computing the hidden state h_t (Hochreiter and Schmidhuber, 1997) . These rules dictate what should be "memorized" and what should be "forgotten" from past steps in the hidden state.

Generative Adversarial Network (GAN)

A *Generative Adversarial Network (GAN)* is trained using a data set. 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* $f_{\boldsymbol{\theta}_d}$. The generator and discriminator can be thought of as two players playing a minimax game. 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 variable given the observed variable. The target variables for GANs are binary labels "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 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. By contrast, 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 Section 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. The 1-Wasserstein distance 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) show 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. There is no known way to enforce the weights of a neural network so it has a certain Lipschitz constant (Arjovsky et al., 2017). Instead, Arjovsky et al. (2017) approximately restrict $f_{\boldsymbol{\theta}_d}$ to 1-Lipschitz functions so the Wasserstein objective approximates the Earth mover's distance. To approximately restrict $f_{\boldsymbol{\theta}_d}$ to 1-Lipschitz functions, Arjovsky et al. (2017) adds a penalty on the difference between the gradient of $f_{\boldsymbol{\theta}_d}$ and 1 (Gulrajani et al., 2017). Recall that 1-Lipschitz functions should have maximum gradient 1 everywhere. The final training objective then becomes

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

where $\hat{\boldsymbol{x}}$ is a random convex combination of an $\boldsymbol{x}$ 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 $\mathbf{z}$ to random graphs $\mathcal{G}$ using a *Variational AutoEncoder (VAE)* (Kingma and Welling, 2013) and trains from a data set $\mathcal{H}^*$ (Simonovsky and Komodakis, 2018). The graphs are represented as adjacency matrices. To obtain permutation invariance, GraphVAE adds an additional graph matching term to the loss which is a log-likelihood of the graphs in $\mathcal{H}^*$ being produced by the VAE. During training, on input $\mathcal{G}^* \in \mathcal{H}^*$ 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 under a permutation that approximately minimizes the loss. The permutation is computed using a quadratic program. The cost of the graph matching 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. (2009). Graph neural networks generate graphs by learning node representations through 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 because its neighbors' representations have been updated to reflect their own neighborhoods.

Li et al. (2018) use graph nets to compute the representation of the current graph. Given a graph net computed graph representation, Li et al. (2018) learn functions to decide whether to (1) add a node and (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. (2018) learn a distribution over graphs by maximizing the likelihood of graphs in $\mathcal{H}^*$ for a fixed node ordering or a node ordering drawn uniformly at random.

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 $\boldsymbol{\phi}$. The transition network $f_{\boldsymbol{\theta}}$ outputs the 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 $t-1$ vector of Bernoulli success probabilities. Graph RNN accommodates graphs of arbitrary size because a graph of size n is constructed after n 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. While the graph is a list of adjacency lists (on the same order of the size of the adjacency matrix),

the RNN itself only has to represent each adjacency list. By side-stepping the need for the RNN 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. Graph RNN training requires that a node ordering 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 absolves 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 that maps graphs to a reward to generate graphs that resemble real-world molecules (De Cao and Kipf, 2018). The loss is a linear combination of the GAN loss, which does not prescribe any features, and the negative of the reward 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 vectors of node types and 3-dimensional tensors containing an edge type for each edge. The node and edge types are important for molecule design because the nodes/edges need to represent different types of atoms/bonds. With the graph representations being size $\Theta(n^2)$, MolGAN does not scale beyond graphs of small size because the neural network also has to be $\Theta(n^2)$. 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 tell apart 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 over graphs that resemble the true samples, incorporating the reward network helps guide the generator towards graphs that share features that are deemed important by the software to identify molecular structure. Thus, these features might be more important than whatever the GAN would discover from discriminator feedback alone.

NetGAN

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 a 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 can train 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 is drawn from the generator and used to construct a graph distribution. A more detailed summary of the NetGAN approach is in Section 5.3 with an in-depth discussion in Chapter 8.

Chapter 5

Our approach toward designing generative graph models around global features

5.1 Overview

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

1. Spectrum-matching generation: Generates graphs by aiming to match the spectrum of the symmetric normalized Laplacian (Chapter 7).

2. Random walk-based generation: Generates graphs by aiming to match the frequencies with which walks are drawn according to some random walk algorithm (Chapter 9).

3. Cut fix generation: Generates graphs by aiming to match 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) to understand how training variants contribute to its success and introduce new variants designed to generate connected output graphs when the input graph is connected (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. In order to match global features imperfectly but well enough 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 can be relaxed.

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, Wilson and Zhu (2008) show that roughly .2% of graphs on 11 vertices have a cospectral mate. We do not know of a method to match the spectrum exactly, but the rareness of cospectral pairs suggests that matching the spectrum approximately is necessary to accommodate the diversity objective discussed in Chapter 3.

For matching random walks and cuts, we choose to match them imperfectly because either identical walk distributions or connectivity across all $2^{\mathcal{V}^*}$ cuts by themselves (or both) imply isomorphism. Identical random walks imply isomorphism because random walks traverse only edges in the graph. We now prove that matching connectivity across all cuts implies isomorphism in Claim 5.1.1.

Claim 5.1.1. *If $\mathcal{G}$ and $\mathcal{G}^*$ have matching connectivity across all cuts, then they are isomorphic.*

Proof. If $\mathcal{G}$ matches the connectivity of all cuts to $\mathcal{G}^*$, this includes all cuts of size 1 and 2. For any two pairs of nodes v and u, the degrees of both nodes can be deduced by the connectivity of $(\{v\}, \mathcal{E}^* \setminus \{v\})$ and the connectivity of $(\{u\}, \mathcal{E}^* \setminus \{u\})$. If the connectivity of $(\{v, u\}, \mathcal{E}^* \setminus \{v, u\})$ is equal to the sum of their degrees, then the two nodes are not connected by an edge. Otherwise, they are connected by an edge. Thus, matching the connectivity of $\mathcal{G}$ to all cuts of size 1 and 2 is sufficient to recover all edges in $\mathcal{G}^*$ and thus $\mathcal{G}$ is isomorphic to $\mathcal{G}^*$. □

More formally, let $\mathcal{G}_n$ be the set of all graphs on $n = n^*$ vertices. One of the main methods we employ across all generative graph models designed in this thesis is constructing the distribution p over $\mathcal{G}_n$ using a probabilistic adjacency matrix A. A probabilistic adjacency matrix A has entries $a_{u,v} \in [0, 1]$ and the distribution p over $\mathcal{G}_n$ is defined using *independent edge sampling* on A (Definition 2.3.1).

5.1.1 Constructing A in iterations

All three generative graph models designed in this thesis share the following high-level framework. The probabilistic adjacency matrix A is constructed over multiple iterations, where each iteration

Model	Update Rule	Importance Heuristic	Utility Score
Spectrum-matching generation	Deterministic rounding across eigencuts	Eigencuts corresponding to small eigenvalues	Connectivity across eigencuts
NetGAN (with new variants presented in Chapter 8)	Stochastic gradient descent on GAN parameters	Bias walks across sparse cuts	Discriminator's ability to tell real walks from fake
Random walk-based generation	Adding mass for nodes that appear multiple times on same walks	Bias random walks to discover sparse cuts	Cuts discovered by walks
Cut fix generation	Adding/removing 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.

makes A more similar to the ground truth adjacency matrix A^*, while (typically) also reducing the entropy. The changes to A are guided by a utility function f, which is different for different models. Starting from an initial matrix $A(0)$ constructed from $\mathcal{G}^*$, in each iteration i, the matrix $A(i-1)$ is updated to $A(i)$. This update is done using f which maps $A(i-1)$ and $\mathcal{G}^*$ to an expected similarity score between $\mathcal{G}^*$ and $\mathcal{G}$ drawn from $p_{A(i-1)}$. The updated matrix $A(i)$ is constructed so that $f(\mathcal{G}^*, A(i)) > f(\mathcal{G}^*, A(i-1))$ in order to make progress on the similarity objective. The entropy score of p_A is $H(p_A) = \sum_{(u,v)\in\mathcal{V}\times\mathcal{V}} h(a_{u,v})/n^2$ where $h(a_{u,v})$ is the entropy of a random Bernoulli variable that takes value 1 with probability $a_{u,v}$ (Section 2.5.2). The update rules for A, importance heuristics on features, and utility scores for each generative graph model are in Table 5.1.

One feature of independent edge sampling is that the spectrum of probabilistic adjacency matrices is a good approximation of the spectra of the output graphs. Chung and Radcliffe show that the spectrum of $\mathcal{G}$ generated by independent edge sampling on A will concentrate around $\boldsymbol{\lambda}(L(A))$ (Theorem 5.1.2). Therefore, progress during iterations to A is tracked using $\ell_2^{\mathrm{LW}}(\boldsymbol{\lambda}(L(A)), \boldsymbol{\lambda}^*)$[1] throughout this thesis to understand how updates to A affect the expected spectrum of $\mathcal{G}$.

Theorem 5.1.2. *[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*

[1]The semimetric ℓ_2^{LW} is an ℓ-2 norm with linear weights inversely proportional to the index; see Section 2.5 for the formal definition.

of the random graph, and $\delta = \min_v \sum_u a_{v,u}$ *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* $L(B)$ *and* $L(A)$ *satisfy* $|\lambda_i(L(B)) - \lambda_i(L(A))| \leq 3\sqrt{\frac{3\ln(4n/\epsilon)}{\delta}}$.

5.1.2 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 5.1.3).

Definition 5.1.1 (Cut connectivity)**.** Let A be a probabilistic adjacency matrix of dimension $n^* \times n^*$ with entries[2] $a_{v,u} \in [0, 1]$. The connectivity of $\mathcal{S} \subseteq \mathcal{V}^*$ denoted by $\partial(\mathcal{S}, A) = \sum_{v \in \mathcal{S}, u \in \overline{\mathcal{S}}} a_{u,v}$ measures the number of edges crossing the cut $(\mathcal{S}, \overline{\mathcal{S}})$.

Definition 5.1.2 (Cut conductance)**.** Let $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ be a graph 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 (fractional) number of edges incident on $\mathcal{S}$. 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 5.1.3 (Graph conductance)**.** Let $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ be a graph with adjacency matrix A^*. The conductance of $\mathcal{G}^*$ is $\Phi(\mathcal{G}^*) = \min_{\mathcal{S} \subseteq \mathcal{V}^*} \varphi_{\mathcal{G}^*}(\mathcal{S})$.

We now discuss how each global feature (the spectrum of the symmetric normalized Laplacian, random walks, and cut connectivity) relates to conductance.

Spectrum of the symmetric normalized Laplacian matrix

The spectrum of the symmetric normalized Laplacian is related to the conductance of a graph by Cheeger's inequality and its higher order variations (Theorems 5.1.3, 7.1.1, 7.1.2).

Theorem 5.1.3. *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,*

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

[2]Cut connectivity is defined the same way for symmetric binary matrices.

The proof of Cheeger's inequality depends on the connectivity across the *Fiedler cut* defined by the eigenvector $\mathbf{x}_2$ that is associated with the *Fiedler value* λ_2. There are multiple ways to define a cut from $\mathbf{x}_2$; for one Fiedler cut definition see Definition 2.2.1.

Cut connectivity

Our notion of cut connectivity is Definition 5.1.1 and is in the definition of conductance. Thus, the two are related.

Random walks

The behavior of random walks is closely tied with conductance. For a Markov chain (including the standard random walk) with transition matrix R and stationary distribution[3] $\boldsymbol{\pi}$, 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 similarly to that of a graph, with $\Phi(R) = \min_{\{\mathcal{S} \subseteq \mathcal{V}^* : \sum_{v \in \mathcal{S}} \pi_v \leq \frac{1}{2}\}} \frac{\sum_{v \in \mathcal{S}, u \in \overline{\mathcal{S}}} r_{v,u}}{\sum_{v \in s} \pi_v}$. Let $r_{v,u}^{(t)}$ denote the v,u-th entry of R^t which denotes the probability of traversing from v to u in the t-th step.

Definition 5.1.4 (Mixing time[4], Theorem 2.2 of Jerrum and Sinclair (1989))**.** The relative pointwise distance $\Delta(t)$ at time t is

$$\Delta(t) = \max_{v \in \mathcal{V}^*, u \in \mathcal{V}^*} \frac{|r_{v,u}^{(t)} - \pi_u|}{\pi_u}.$$

With $\pi_{\min} = \min_{i \in \mathcal{V}^*} \pi_v$, if[5] $\min_v r_{v,v} \geq \frac{1}{2}$ then

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

.

Theorem 5.1.4 says that the largest deviation in the distribution over states at time t from the stationary distribution is bounded by $\Delta(t) \leq \frac{((1-\Phi(R)^2)/2)^t}{\pi_{min}}$. This implies that if the conductance of R is large, then the walks will mix quickly. If the conductance of R is small, we need another result

[3] Random walk notation is in Section 2.4.

[4] This mixing time result assumes that the Markov chain R is time-reversible and irreducible. Irreducibility is defined in Section 2.4; for the definition of time-reversibility, see Jerrum and Sinclair (1989).

[5] This condition imposes a staying probability of at least $\frac{1}{2}$ at each vertex. This is similar to the "lazy" random walk condition imposed in Proposition 5.1.1.

to show that the walks will mix slowly. 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 on the $\mathcal{S}$ side of the cut $(\mathcal{S}, \overline{\mathcal{S}})$, 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 5.1.1 (Time it takes walks to leave cuts, Proposition 2.5 (Spielman and Teng, 2013))**.** *The probability a walk with transition matrix $R = \frac{1}{2}(A^*D^{-1} - I_{n^*})$ on $\mathcal{G}^*$ starting in $\mathcal{S} \subseteq \mathcal{V}^*$ of length t stays entirely in $\mathcal{S}$ is at least $1 - t\varphi_{\mathcal{G}^*}(\mathcal{S})/2$.*

5.2 Spectrum-matching generation

Spectrum-matching generation aims to sample from the class of all graphs with similar spectra (Shine and Kempe, 2019) (Chapter 7). Cheeger's inequality and variants show that approximately matching spectra is related to approximately matching the connectivity across sparse cuts that avoid overlap (Theorems 5.1.3, 7.1.1, and 7.1.2).

Concretely, take 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 the characterization of the symmetric normalized Laplacian $L = W_{\boldsymbol{\lambda}^*}(X) = X\Lambda^* X'$ where X is an orthonormal eigenbasis and Λ^* 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 Spectrum-matching 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 $W_{\boldsymbol{\lambda}^*}(X)$. This step is detailed in Section 7.3.

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. This step is detailed in Section 7.4.

3. Use an LP-based algorithm to minimally perturb the entries of $\tilde{A}$ by a matrix E to construct the 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. This step is detailed in Section 7.5.

4. Keeping within the general framework in Section 5.1.1, iterate on $\tilde{A}$ in a series of rounds to better match connectivity across eigencuts with small eigenvalues using deterministic rounding followed by independent edge sampling (Definition 2.3.1). This step is detailed in Section 7.6.

5.3 NetGAN with new variations that target sparse cuts

In recent work, Bojchevski et al. (2018) proposed NetGAN, 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 neural network that samples from distributions (Section 4.5.1). In NetGAN, 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[6].

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. If each additional training step successfully updates the GAN generator to produce "walks" that better resemble the real walks, then with each additional training step the frequency matrix $\tilde{A}$ should also have entries that are more proportional to those in A^*.

4. Use an iterative sampling procedure to produce sample graphs $\mathcal{G}$ from $\tilde{A}$. The procedure samples exactly m^* edges 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.

Bojchevski et al. (2018) show that the graphs $\mathcal{G}$ produced by NetGAN perform well on edge prediction with regards to: (1) area under the receiver operating characteristic curve (AUC), and

[6]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, consecutive integers need not represent an edge in any graph.

(2) average precision (AP) (Powers, 2011). (Definitions of AUC and AP are in Section 8.1.4.) It is also shown that the graphs perform competitively with other graph generative models with respect to matching several graph features.

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

Observation 1. *$\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ and the choice of random walk define a probability distribution 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) use the standard random walk and a generalization called *node2vec* walk described in Section 8.1. We introduce training with an alternative 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. This is discussed further in Section 8.3.

- The length of random walks used. As stated in Observation 1, any information encoded about the order the nodes appear in the walks is lost in the projection from walks to $\tilde{A}$. What is the role of having walks instead of edges? This is discussed further in Section 8.4.

- The stopping criterion for stopping GAN training early. Bojchevski et al. (2018) stop GAN training before the GAN learns the walk distribution exactly and memorizes A^*. We incorporate global structure directly into the stopping criterion to help ensure $\tilde{A}$ has seen enough walks to learn the global structure of the graph. This is discussed further in 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 analyze an independent edge sampling

on a scaled version of $\tilde{A}$ instead of the iterative procedure used by Bojchevski et al. (2018). This is discussed further in Section 8.5.

5.4 Random walk-based 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. To avoid this opaque mapping, we propose in this thesis a different model that we call Random walk-based generation. Random walk-based generation works by mapping 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-based generation expands the set of pairs that it considers as candidates for edges. Instead of considering only pairs that appear on the walks consecutively along edges, our approach considers all node pairs that appear on the same walk. The idea is that the more often any two nodes appear on the same walk, the more likely the two nodes are connected by a short path in a dense area. By counting node pair frequencies instead of edges, the goal is to keep dense areas in $\mathcal{G}^*$ dense in $\mathcal{G}$ without memorizing edges so that the input graphs are diverse. Node pair frequencies not only help us capture the dense areas, but sparse cuts as well. If two nodes v and u rarely appear on the same walk, then they are likely separated by a sparse cut. Thus, including edge (v, u) with probability proportional to the number of times v and u appear on the same walk helps keep cuts that separate them sparse.

Random walk-based generation takes place in rounds (keeping with the high-level framework described in Section 5.1.1). Each round has a cut construction phase and a phase in which $\tilde{A}$ is updated 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 $\tilde{a}_{v,u}$ based on the number of times v and u appeared on the same walk together.

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

5.5 Cut fix generation

All of the above methods aim to match the connectivity across sparse cuts indirectly by matching eigenvalues or random walk behavior, both of which are related to the connectivity across sparse cuts. We propose a new generative graph model that we call *Cut fix generation* that can start with any seed probababilistic adjacency matrix A. It then iteratively adds/removes edges across cuts for which the discrepancy from $\mathcal{G}^*$ is largest (Chapter 10). Keeping with the high-level framework in Section 5.1.1, Cut fix generation is performed in rounds where $A(i-1)$ is updated to $A(i)$ in the i-th round with $A(0)$ equal to the seed matrix A. In the i-th round there are two phases:

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

Once enough rounds have been completed, $\mathcal{G}$ is constructed by performing independent edge sampling on A (Definition 2.3.1).

5.6 Evaluation

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

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

Stochastic block model (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 model (Kron) The Kronecker model by Leskovec et al. (2010) 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.

5.7 Choosing an appropriate generative graph model

When choosing what type of generative graph models to use, there are three important factors to consider:

1. The desired application of the output graphs and if there are any graph features that need to be shared among the graphs.

2. The number of input graphs available to build the model from.

3. How long it takes to build the model.

5.7.1 Features of interest

For many of the applications discussed in Chapter 1 (like simulation studies or robustness tests), there are graph features that are intrinsic to the intended use of the generated graphs. For these, it is important to choose a generative graph model that samples from graphs with those features of interest. If there is a prescriptive generative graph model that uses these features of interest, this would be a clear choice over a dynamic process or data driven model that does not match those features directly. For an example of an application and a graph feature of interest, one clear use for the models described in this thesis is generating synthetic data for simulations of contagion processes that depend on the global structure matched by these models. (Further discussion in Section 1.1.1.)

Alternatively, other applications are more exploratory and it is not clear what features are desired to be among the generated graphs. For example, in Section 1.1.2 we discuss the generative graph models for generating molecules to study how they interact. For these applications, GraphVAE which is discussed in Section 4.5.2 could be a better choice because it does not require features to be prescribed.

5.7.2 Data available

Another aspect to consider is what type of data is available. If there are many graphs, then data driven approaches are available that train from a data set of graphs (for example, GraphRNN discussed in Section 4.5.2). However, if there is only one graph to build the model from, then a model designed to take a single graph as input (like those presented in this thesis) would be a better fit. Sometimes there are not any graphs to train from, only a desired real-world phenomena that builds networks. In this case, dynamic process models in the vein of those discussed in Section 4.4 would be the clear choice.

5.7.3 Time constraints

Of the three new generative graph models presented in this thesis and the NetGAN variants introduced, Cut fix generation is the fastest and has scaled to graphs as large as 10,000 nodes (Section 10.4.3). Our experiments suggest that the size of the input graph (usually the number of nodes) is a key indicator for how long building the models in this thesis will take. On the basis of time, Cut fix generation is a good choice from the generative graph models presented in this thesis for generating large graphs that share global structure.

However, the Configuration model and Stochastic Block model are simple to fit and are much faster then our methods (including Cut fix generation). For all the inputs we tried, the Configuration model and Stochastic block model took seconds whereas the Cut fix generation method takes minutes once graphs are on the order of a few hundred. Therefore, if global structure is not relevant to the task at hand then one of the existing simple models would be a better choice.

Chapter 6

Datasets

6.1 Real-world datasets

We use a number of real-world graphs as input graphs for our evaluation. All of these graphs 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. **Social networks**: The nodes are "social" entities and an edge exists between two nodes if they share a "social" connection. Our experiments use unweighted graphs, so information regarding the strength of the connection is omitted. For example, in the Email network the number of emails shared between two people is ignored.

   (a) Football, NCAA college football 2000. A network of football games between colleges during a single season where each node is a team and an edge is between two teams if they played each other during that season.

   (b) Netscience, Scientific collaborations in network science (2006). A network of collaborations between authors of network science papers where the nodes are scientists and an edge exists if the two scientists coauthored at least one paper.

   (c) Email, Email network (University R-V, Spain, 2003). An email exchange network between members of a university where an edge exists between two university members if at least one email was sent from one to the other.

(d) HEALTH, Adolescent health (1994). A network of friendships obtained through a health survey. Each node is a high-school student. Each student was asked to name at most 5 female and 5 male friends. An edge exists between two students if at least one named the other in their friend list.

(e) ADVOGATO, Advogato trust network (2009). A network of trust relationships among users on Advogato, an online community of open source software developers. Trust among the developers is self-reported and an edge exists between two developers if at least one reported that they trusted the other.

(f) HEPHYSICS, Scientific collaborations in physics (1995-2005). Collaboration graph for high-energy physicists. The nodes are scientists and an edge exists between two nodes if the scientists have coauthored at least one paper.

2. **Transportation networks**: The nodes are locations and an edge exists between two locations if there is a way to transport directly from one to the other.

(a) AIRPORT, US airport network (top 500; 2002). A network of flights between the busiest 500 commercial airports in the US measured by the number of flights flying in/out of the airport. Each node is an airport and two nodes are connected by an edge if there was a flight between the two airports.

(b) EUROROAD, Euroroad network (2011). A network of international "E-roads", mostly in Europe; the sampling method is unclear.

(c) ROME, Rome roads (1999). A network of roads in Rome. The nodes are intersections between roads and edges are roads or road segments.

3. **Web network**: The nodes are web pages and an edge exists between two nodes if the web pages hyperlink each other.

(a) WIKI, Wikipedia norms (2015). Links between Wikipedia pages on editorial norms. Wikipedia pages on editorial norms include pages on content creation and interactions between users. The nodes are pages and an edge exists if at least one of the pages hyperlinks to the other.

(b) Amazon, Amazon pages (2012). A small sample of web pages from Amazon.com and its sister companies; the sampling method is unclear.

6.2 Synthetic data

To test the ability of the generative graph models to approximate the structure of 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 Erdős-Rényi $G(n, p)$ 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 (30 by 30) with 900 nodes. The grid does not include diagonals.

6.3 Basic parameters of graphs

The basic parameters of the real-world input social (Table 6.1), transportation (Table 6.2), web (Table 6.3) and synthetic graphs (Table 6.4) of the largest connected component are summarized in the associated Tables.

	Number of Vertices	Number of Edges	Spectral Gap	Fiedler Cut Conductance
Football	115	613	0.1368	0.1077
Netscience	379	914	0.003	0.0048
Email	1133	5451	0.1211	0.1493
Health	2539	10455	0.0379	0.0512
Advogato	5042	40509	0.0674	0.0909
Hephysics	11204	117619	0.0018	0.0024

Table 6.1: Basic properties of the social input graphs.

	Number of Vertices	Number of Edges	Spectral Gap	Fiedler Cut Conductance
Airport	500	2980	0.0274	0.0588
Euro road	1039	1305	0.0005	0.0063
Rome	3353	4831	0.001	0.011

Table 6.2: Basic properties of the transportation input graphs.

	Number of Vertices	Number of Edges	Spectral Gap	Fiedler Cut Conductance
Wiki	1872	15367	0.1295	0.175
Amazon	2879	3886	0.0686	0.0933

Table 6.3: Basic properties of the web input graphs.

	Number of Vertices	Number of Edges	Spectral Gap	Fiedler Cut Conductance
Five Cluster	500	12381	0.0001	0.0001
Grid 900	900	1740	0.0029	0.0252

Table 6.4: Basic properties of the synthetic input graphs.

Chapter 7

Generating graphs subject to Laplacian spectra

Spectrum-matching 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 $\boldsymbol{\lambda}(L(B)) = \boldsymbol{\lambda}$.

7.1 Overview

One pivotal result relating graph connectivity to Laplacian spectra is *Cheeger's inequality* (Theorem 5.1.3) which relates graph conductance (Definition 5.1.3) to the second-smallest eigenvalue λ_2 which is equal to 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 $k \leq n$ there are k cuts with conductance characterized by λ_k.

Theorem 7.1.1 (Theorem 1.1 of (Lee et al., 2014))**.** *Let $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ be a d-regular graph with $d_v = d$ for all $v \in \mathcal{V}^*$ and $2 \leq k \leq n$. The conductance of $\mathcal{S} \subset \mathcal{V}^*$ is written as $\varphi_{\mathcal{G}^*}(\mathcal{S})$ (Definition 5.1.2). Let*

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

[1]The second-smallest eigenvalue is called the spectral gap because the smallest eigenvalue of the symmetric normalized Laplacian is zero for all graphs, so λ_2 measures the gap between the smallest eigenvalue and itself.

where $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k$ ranges over all partitions of $\mathcal{V}^*$ into k parts. Then,

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

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

Theorem 7.1.2 (Theorem 1.2 of (Lee et al., 2014))**.** *Let $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$ be a d-regular graph with $d_v = d$ for all $v \in \mathcal{V}^*$ and $2 \leq k \leq n$. 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 $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k$ ranges over all partitions of $\mathcal{V}^*$ into k parts. Then,

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

Theorems 7.1.1 and 7.1.2 imply that if the smallest k eigenvalues of two d-regular graphs are close, then there exist k cuts in both graphs with similar connectivity. While in general we study $\mathcal{G}^*$ 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 $\mathcal{G}^*$.

At a high level, Spectrum-matching 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 7.3).

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 7.4).

3. Use an LP-based algorithm to minimally perturb the entries of $\tilde{A}$ by a matrix E to construct a probababilistic 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 7.5),

4. Use a combination of deterministic rounding across eigencuts with small eigenvalues and independent edge sampling (Definition 2.3.1) of other edges to round A to B which produces the output graph $\mathcal{G}$ (Section 7.6).

Note that our goal is to to find a $\mathcal{G}$ that matches the spectrum of $\mathcal{G}^*$ approximately, not exactly. We relax the problem to matching the spectrum because simulation studies suggest that even 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).

7.2 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 (1982) present a method for generating non-isomorphic co-spectral graphs with respect to the adjacency matrix. There has also been work on generating certain families of co-spectral graphs with respect to the Laplacian matrix (Merris, 1997). In this context, 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 (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

[2]The similarity and diversity objectives are described in Chapter 3.

Wilson, 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, much of our effort in the design of Spectrum-matching generation is focused on actually producing a suitable adjacency matrix, and on choosing a "Laplacian matrix" that lends itself to this transformation (Section 7.3). As far as we can tell, Xiao and Hancock (2006) do not describe any such procedure. White and Wilson (2007) suggest thresholding the values of the matrix, including as edges of the graph those pairs (v, u) for which the entries of the Laplacian are most negative. We compare two different thresholding approaches in this vein to our algorithms in Section 7.7, and find that they perform significantly inferior in matching the spectrum of the input graph.

7.3 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 Λ^* is the diagonal matrix with $\boldsymbol{\lambda}^*$ on the diagonal. An orthonormal basis is an $n^* \times n^*$ 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 $L(B) = I_{n^*} - D(B)^{-1}BD(B)^{-1} = W_{\boldsymbol{\lambda}^*}(X)$ where $D(B)$ is the diagonal degree matrix of B with row sums on the diagonal so that the entry $d(B)_{v,v} = \sum_{u \in \mathcal{V}^*} b_{v,u}$. 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, using Linear Programming we find a symmetric template matrix $\tilde{A}$ such that $L(\tilde{A}) = W_{\boldsymbol{\lambda}^*}(X)$ where $\tilde{A}$ is (likely) not binary.

Our linear program builds from the following observation about symmetric normalized Laplacians

(Lemma 7.3.1).

Lemma 7.3.1. *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. First we show that if the matrix Z is the Laplacian of some symmetric matrix $\tilde{A}$ with positive row sums, then there exists a positive vector $\mathbf{y}$ such that $Z\mathbf{y} = 0$. By assumption, $Z = L(\tilde{A}) = I - D(\tilde{A})^{-1/2}\tilde{A}D(\tilde{A})^{-1/2}$ 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 $Y = \mathrm{diag}(y_1, y_2, \ldots, y_n)$, 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 $\tilde{A}$. □

The vector entries y_v in Lemma 7.3.1 are the square roots of the degrees of nodes v. For basis X, it 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 $\mathbf{y} > \mathbf{0}$.

$$\begin{aligned} \text{Minimize} \quad & b & & (7.1) \\ \text{subject to} \quad & |\sum_{u \in \mathcal{V}^*} z_{v,u} y_u| \leq b & v \in \mathcal{V}^* & (7.2) \\ & y_v \geq \epsilon & v \in \mathcal{V}^*. & (7.3) \end{aligned}$$

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

Algorithm 1: SpectralFitting

Result: $\tilde{A}$: matrix with $\boldsymbol{\lambda}(L(\tilde{A})) \approx \boldsymbol{\lambda}^*$

Input: X: orthonormal matrix ;

$\boldsymbol{\lambda}^*$: vector of desired eigenvalues ;

m^*: number of desired edges ;

ϵ: constant to impose positive row sums ;

$Z = W_{\boldsymbol{\lambda}^*}(X)$;

Solve LP (7.1) with ϵ, 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 $b = 0$, then $L(\tilde{A}) = W_{\boldsymbol{\lambda}^*}(X)$ and the template matrix $\tilde{A}$ output by Algorithm 1 has exactly the desired spectrum $\boldsymbol{\lambda}^*$. We note that the spectrum is invariant to the scaling in the 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 $b > 0$, 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 b; the lemma also motivates our choice of LP objective.

Lemma 7.3.2. *Let δ be such that $b \leq \delta y_i$ for all $i = 1, 2, \ldots, n^*$. Then, the perturbed eigenvalues satisfy $|\lambda_i(L(\tilde{A})) - \lambda_i(Z)| \leq \frac{\delta}{1-\delta} \cdot |\lambda_i(Z)|$.*

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 $Q = \mathrm{diag}(q_1, q_2, \ldots, q_{n^*})$, the perturbed Laplacian satisfies $L = QZQ'$.

Applying Theorem 7.3.3 (below), we get that the relative perturbation in eigenvalues is at most

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

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

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

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

Theorem 7.3.3 (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, $|\tilde{\lambda}_i - \lambda_i| \le |\lambda_i| \cdot ||Q'Q - I||_2$, for all i.*

7.4 Stiefel Manifold Optimization

While the techniques in Section 7.3 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 $\mathcal{S}_{n^*} = \{X \;:\; X'X = I_{n^*}\}$, 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 $Y_X = \text{diag}(\mathbf{y}_X)$. 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 $-\text{grad}(F(\tilde{A}(X_t)))$ 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 $\text{grad}(F(\tilde{A}(X_t)))$, 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 these four steps 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.

7.5 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). The sum $d_v = \sum_u \tilde{a}_{v,u}$ denotes the (fractional) degree of i.

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

$$\text{subject to} \sum_{u \in \mathcal{V}^*} e_{v,u} = 0, \qquad v \in \mathcal{V}^* \tag{7.5}$$

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

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

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

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

Proof. Consider the perturbation $\Delta = L(\tilde{A}) - L(A)$ of the Laplacian matrix. Applying Theorem 7.5.2 (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 $\sum_i |\lambda_i(\Delta)| \leq b$.

Because Δ is symmetric and real-valued, $\Delta = U'\Lambda U$ 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 $\hat{\Delta} = U'\hat{\Lambda}U$. 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, $\hat{\Delta} = U'\Sigma U\Delta = W\Delta$, where $W = U'\Sigma U$ is a unitary matrix. Because W is unitary, its entries are bounded by 1 in absolute value. We can therefore bound

$$\sum_i |\lambda_i(\Delta)| \;=\; Tr(\hat{\Delta}) \;=\; Tr(W\Delta) \;=\; \sum_v \sum_u w_{v,u}\delta_{u,v} \;\leq\; \sum_{v,u} |\delta_{u,v}|.$$

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 b. □

Theorem 7.5.2 (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 $\sum_i \Phi(\lambda_i(A) - \lambda_i(B)) \leq \sum_i \Phi(\lambda_i(B - A))$.*

7.6 Matrix Rounding

Theorems 7.1.1 and 7.1.2 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 the sign of the entries[6] in $\mathbf{x}$ (Definition 7.6.1).

[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 using signs for the sake of computational time.

Definition 7.6.1. *Eigencut with signs.* The eigencut $\mathcal{S} \subseteq \mathcal{V}^*$ associated with the eigenvector $\mathbf{x}$ of L is the set $\{v \ : \ \text{sign}(x_v) = \psi\}$ where ψ takes $+1$ or -1 according to which makes the set have smaller volume.

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 $\mathcal{S}_k$ 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 7.6.2).

Definition 7.6.2. *Critical cut.* Let A be an $n^* \times n^*$ 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 7.6.1) associated with $\mathbf{x}_k$ where k is the smallest integer such that $|\mathcal{S}_k| \geq \frac{1}{2}|\mathcal{S}^*|$.

Recall that $d_v = \sum_u a_{v,u}$. 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 λ_k is too large or too small[9]. 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.

[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.

[9] Experimentally, we find λ_k is usually too large so we focus on that case.

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.

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). Scores are assigned 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 $\mathbf{x}$, critical cut eigenvector $\mathbf{x}'$, and scores are recomputed on A. In addition, if the (fractional) number of edges crossing the Fiedler cut is below 1, the procedure is stopped. To keep the overall edge density reasonably constant, the algorithm alternates 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 $\lambda_k(L(A))$ and added to the approximate score total r to estimate how close $\lambda_k(L(A))$ is to matching λ_k^* after each alteration. An iteration terminates once the approximation 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

Algorithm 2: Critical Cut round

Result: A: Matrix with entries $a_{v,u} \in [0,1]$ and $\lambda_k(L(A)) \approx \lambda_k^*$
Input: A: Matrix with entries $a_{v,u} \in [0,1]$;
$\boldsymbol{\lambda}^*$: desired spectrum ;
z: budget of mass that can be removed from edges crossing cuts ;
c: closeness threshold ;

Compute critical cut $\mathcal{S}_k$ associated with eigenvector $\mathbf{x}'$ (Definition 7.6.2) and Fiedler cut $\mathcal{S}^*$;
Difference $\delta = |\lambda_k(L(A)) - \lambda_k^*|$;
Set unchanged flag to False ;
repeat
  Compute lists of pairs of nodes removalCandidates and additionCandidates and their scores $f(v,u)$ (Algorithm 3) ;
  Round entries $a_{v,u}$ down/up to 0/1 for pairs (v,u) 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 $a_{v,u}$ that have been rounded down across the cut ;
  Initialize $r = 0$, which keeps track of the sum of scores of altered pairs (v,u) 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 remain candidates in removalCandidates and additionCandidates to be processed* **do**
    if $y \geq 0$ **then**
      (v,u) is the pair (v',u') in removalCandidates with the largest score. Discard (v,u) from removalCandidates. Mark (v,u) to be removed ;
    else
      (v,u) is the pair (v',u') in additionCandidates with the largest score. Discard (v,u) from additionCandidates. Mark (v,u) to be added.
    if $r + f(v,u) \leq \delta$ **then**
      Increment r by $f(v,u)$;
      if (v,u) *to be removed* **then**
        if (v,u) *is not the last removal candidate* **then**
          Increment q by $a_{v,u}$;
          Decrement y by $a_{v,u}$;
          Set $a_{v,u}$ to 0 ;
      else
        Increment y by $1 - a_{v,u}$;
        Set $a_{v,u}$ to 1 ;
  end
  Compute critical cut $\mathcal{S}_k$ associated with eigenvector $\mathbf{x}'$ (Definition 7.6.2) ;
  If A is unchanged, set unchanged to True ;
until $|\lambda_k(L(A) - \lambda_k^*| < c$, *the sum of entries crossing $\mathcal{S}_k$ or $\mathcal{S}^*$ is less than 1, or unchanged is True*;

Algorithm 3: Compute addition/removal candidates

Result: Lists of pairs of nodes removalCandidates and additionCandidates ;
Input: A: matrix with entries $a_{v,u} \in [0,1]$;
λ^*: desired spectrum ;

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 (v,u) to removalCandidates.
  else
    Score $f(v,u) = \frac{x'_v x'_u (1-a_{v,u})}{\sqrt{d_v d_u}}$. Add (v,u) to additionCandidates.

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. 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 makes the sum of fractional edges crossing the Fiedler cut less than one, the graph is 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 independent edge sampling (Definition 2.3.1) to round any remaining non-binary entries $a_{v,u}$. Theorem 5.1.2 bounds the resulting spectral perturbations (Chung and Radcliffe, 2011).

We also experimented with using a dependent rounding scheme due to 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 in our final algorithm.

Algorithm 4: Critical Round

Result: A: matrix with entries $a_{v,u} \in [0,1]$ and $\lambda_k(L(A)) \approx \lambda_k^*$
Input: A: matrix with entries $a_{v,u} \in [0,1]$;
λ^*: desired spectrum ;
z: budget of mass that can be removed from edges crossing cuts ;
c: closeness threshold ;

Set cutCorrected flag to False ;
repeat
  Consider the graph of edges (u,v) with $a_{u,v} > 0$. ;
  if *Graph is disconnected* **then**
    Redefine A on the entries of the largest connected component ;
  Compute Fiedler cut $\mathcal{S}^*$;
  while $|\mathcal{S}^*| < 5$ **do**
    Remove $\mathcal{S}^*$ and recompute Fiedler cut $\mathcal{S}^*$;
  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 *Sum of entries crossing the Fiedler cut is less than 1* **then**
  Round the largest entry crossing up to 1 ;

7.7 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 7.6, 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.

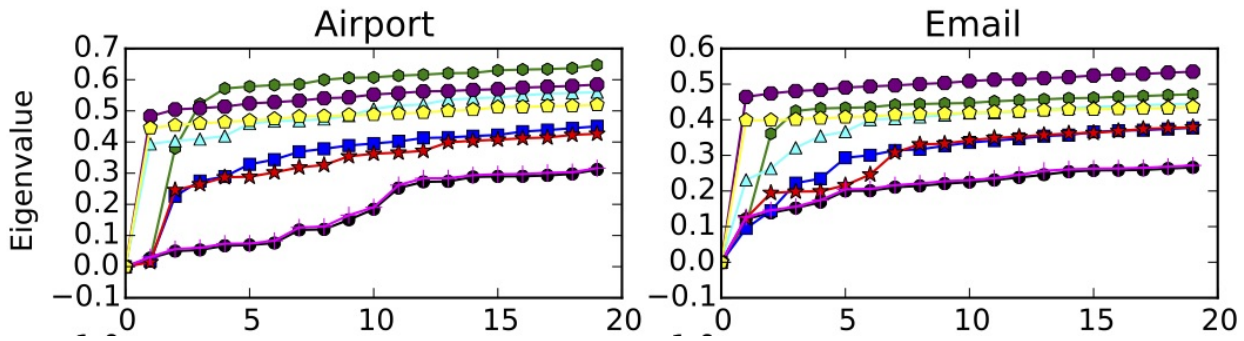


Figure 7.1: Spectra of matrices produced by employing different sets of heuristics. Input graph (black circles), rounding an unfitted matrix (green hexagons), fitted template matrix (magenta plus), fractional graph (cyan triangles), output graph without Stiefel manifold optimization (blue squares), output 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.

The experimental results comparing Spectrum-matching generation to the other graph generation methods are in Chapter 11.

Chapter 8

Generating graphs with deep random walks

The behavior of random walks is determined by the connectivity structure of the graph (Section 5.1.2)[1]. For example, how many steps it takes for a walk to leave a set of nodes tells us something about how connected the set of nodes is to the rest of the graph (Spielman and Teng, 2013). This chapter explores how to build random graphs from random walks of a fixed length drawn from target graph $\mathcal{G}^*$ 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 $\mathcal{G}$ with probability proportional to these counts. 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 construct $\mathcal{G}$ using the real walks, only edges in $\mathcal{G}^*$ could appear in $\mathcal{G}$ and $\mathcal{G}$ would be a subgraph of $\mathcal{G}^*$. Therefore, the GAN acts as a *noise generator* to introduce variation into $\mathcal{G}$.

We investigate how NetGAN variants can be chosen to target learning the global 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 in whether global properties can be learned under this paradigm where we learn a distribution over semi-local

[1] Random walks are defined in Section 2.4.
[2] Generative adversarial networks are defined in Section 4.5.1.

features and then combine these features 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) show that 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 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 the deep models 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 that are specific to only a few graphs. 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 $\mathcal{G}^*$. One 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 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 are exponential in the number of vertices?

3. A method to construct a 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) design *NetGAN* and are 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 symmetric 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 walk. 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 with probability proportional to $\tilde{a}_{v,u}$. Algorithm 5 describes NetGAN in additional detail.

One motivation for training the GAN with random walks instead of 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 take as input and reduces the problem to creating a node representation. A node representation 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

[3]A discussion about the difference between local and global features is in Section 1.3.

enough, but not perfectly, should result in implicitly learning a fairly accurate representation of the 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 exclusively on edge deletions to diversify the samples, NetGAN uses an early stopping criterion to stop training before the GAN finishes learning p^*. By stopping training early, the learned distribution traverses pairs $(v, u) \notin \mathcal{E}^*$.

In the first phase of the NetGAN generation algorithm, integer sequences $(v_1, v_2, \ldots, v_k)$ where $v_t \in \mathcal{V}^*$ are generated from a partially trained GAN. These integer sequences 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. This is explained further in Section 8.1.3. The second phase of the NetGAN algorithm uses synthetic walks from the GAN generator to construct $\mathcal{G}$.

The GAN is trained to learn "random walk" sequences[4] that are similar to random walks on $\mathcal{G}^*$. To detect overfitting, the random walk data the GAN is trained using a subset of $\mathcal{E}^*$ and a disjoint hold-out set is used to discern if the GAN is memorizing. The edge set $\mathcal{E}^*$ is split into three parts: a training set $\mathcal{E}_r$, a validation set $\mathcal{E}_v$ and a 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). Both RNNs are a special type called a *Long-Short Term Memory* machine (Hochreiter and Schmidhuber, 1997). The RNN memory state and input at time t are denoted by $\boldsymbol{m}^{(t)}$ and $\boldsymbol{u}^{(t)}$ respectively. To see how the generator RNN is used to generate integer

[4]Random walks, as defined in Section 2.4, are defined on graphs so consecutive pairs of vertices appear as edges in a graph. The GAN generates integers that represent vertices; however, consecutive integers need not represent an edge in any graph.

Algorithm 5: NetGAN algorithm.

Result: $\mathcal{G}$
Input: $\mathcal{G}^*$;
$\mathcal{A}$: random walk algorithm ;
k: random walk length ;
T: number of evaluation transitions ;
b: batch size ;
Hold out set parameters: validation set size $0 < q_v < 1$;
test set size $0 < q_t < 1 - q_v$;
stoppingCondition: condition to stop training ;
ℓ: number of training steps before evaluation ;
$\tilde{B}$: utility function on frequency matrices ;

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 ℓ 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**
        $\boldsymbol{w} = \mathcal{W}_i$;
        for $j \leftarrow 1$ **to** k- *1* **do**
            $u = \boldsymbol{w}[j]$, $v = \boldsymbol{w}[j+1]$;
            $\tilde{b}_{v,u} = \tilde{b}_{v,u} + 1$;
            $\tilde{b}_{u,v} = \tilde{b}_{u,v} + 1$;
        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	Basis vector $\boldsymbol{e}_v^{n^*}$	Vertex $v \in \mathcal{V}^* = [n^*]$
State	Vector $\boldsymbol{v}^{(t)} \in \{\boldsymbol{e}_i^{n^*}\}$	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.

sequences that can be interpreted as random walks, $\boldsymbol{p}^{(t)}$ is an 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.

The state $\boldsymbol{v}^{(t)}$ is encoded as a basis vector $\boldsymbol{e}_v^{n^*}$. The vertex set is $\mathcal{V}^* = [n^*]$ so there is a correspondence between node $v \in \mathcal{V}^*$ and $\boldsymbol{e}_v^{n^*}$. To help accommodate large n^*, $\boldsymbol{v}^{(t)}$ is projected to an n'-dimensional input vector $\boldsymbol{u}^{(t)}$ with $n' < n^*$ using a projection function $\hat{g}_{\boldsymbol{\gamma}_g}(\boldsymbol{v}^{(t)})$ where $\boldsymbol{\gamma}_g$ is learned. The RNN generator is written as $g_{\boldsymbol{\theta}_g}(\boldsymbol{m}^{(t)}, \boldsymbol{u}^{(t)})$ where $\boldsymbol{\theta}_g$ is a learned vector. The state $\boldsymbol{v}^{(t)}$ takes value $\boldsymbol{e}_v^{n^*}$ for $v \in \mathcal{V}^*$ with probability $p_v^{(t)}/Z$ where $Z = \sum_{v \in \mathcal{V}^*} p_v^{(t)}$. The initial $\boldsymbol{m}^{(0)}$ is sampled by mapping a random Gaussian noise vector to an s_g-dimensional vector. The parameters $\boldsymbol{\nu}_g$ of the noise mapping $\hat{\hat{g}}_{\boldsymbol{\nu}_g}$ are also learned. At time step t, $g_{\boldsymbol{\theta}_g}(\boldsymbol{m}^{(t-1)}, \boldsymbol{u}^{(t-1)}) = (\boldsymbol{m}^{(t)}, \boldsymbol{p}^{(t)})$ and $\boldsymbol{v}^{(t)}$ is sampled according to $\boldsymbol{p}^{(t)}$. After k calls, the RNN has produced 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 $\boldsymbol{\theta}_d$. At each time step t, the discriminator (1) reads in the basis vector $\boldsymbol{v}^{(t)}$ and projects it down to $\hat{f}_{\boldsymbol{\gamma}_d}(\boldsymbol{v}^{(t)}) = \boldsymbol{u}^{(t)}$ (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 $o^{(k)} = \hat{\hat{f}}_{\boldsymbol{\eta}_d}(\boldsymbol{o}^{(k)})$ 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 $\boldsymbol{w} = v_1, v_2, \ldots, v_k$: $F_{\hat{\boldsymbol{\theta}}_d}(\boldsymbol{w}) = o^{(k)}$. The 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 $\mathcal{W} = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_b\}$ are generated from the actual random walk process, and another batch of b "fake" walks $\mathcal{W}(\hat{\boldsymbol{\theta}}_g) = \boldsymbol{w}'_1, \ldots, \boldsymbol{w}'_b$ are generated by the generator (Algorithms 7,6). The average output on real/fake walks is $a(\mathcal{W}, \hat{\boldsymbol{\theta}}_d) = \frac{1}{b}\sum_{i=1}^{b} F_{\hat{\boldsymbol{\theta}}_d}(\boldsymbol{w}_i)$, $a(\mathcal{W}(\hat{\boldsymbol{\theta}}_g), \hat{\boldsymbol{\theta}}_d) = \frac{1}{b}\sum_{i=1}^{b} F_{\hat{\boldsymbol{\theta}}_d}(\boldsymbol{w}'_i)$, 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. The parameters $\hat{\boldsymbol{\theta}}_g$ and $\hat{\boldsymbol{\theta}}_d$ are updated using stochastic batch gradient descent; the details are in Algorithm 8.

Algorithm 6: GenerateRealWalks

Result: Walks drawn from $\mathcal{G}_r$ using $\mathcal{A}$: $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_b$;
Input: b: number of walks ;
k: walk length ;
$\mathcal{A}$: walk algorithm ;
$\mathcal{G}_r$: training graph ;

Draw b real random walks $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_b$ 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 $\hat{\boldsymbol{\theta}}_g$: $\boldsymbol{w}'_1, \boldsymbol{w}'_2, \ldots, \boldsymbol{w}'_b$
Input: $\hat{\boldsymbol{\theta}}_g$: generator parameters ;
b: number of walks ;
k: length of walks ;

Draw b fake random walks $\boldsymbol{w}'_1, \boldsymbol{w}'_2, \ldots, \boldsymbol{w}'_b$ 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

[5] Wasserstein loss is 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: $\hat{\boldsymbol{\theta}}_g$: generator parameters ;
$\hat{\boldsymbol{\theta}}_d$: discriminator parameters ;
ℓ_d: number of discriminator iterations ;
ℓ_g: number of generator iterations ;
b: number of walks ;
k: length of walk ;
$\mathcal{A}$: random walk algorithm ;
$\mathcal{G}_r$: training graph ;

---

for $i \leftarrow 1$ **to** ℓ_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** ℓ_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

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 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.

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: $\mathcal{G}_r$: training graph ;
b: number of walks ;
k: length of walks ;
Q: transition matrix ;

π 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 π_u ;
  for $j \leftarrow 2$ **to** k **do**
    $\boldsymbol{w}_i[j] = u$ with probability $q_{u,\boldsymbol{w}_i[j-1]}$;
  end
end

Bojchevski et al. (2018) use 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 and the probability of traversing from v to u is $\frac{1}{2m^*}$ for all $v, u \in \mathcal{V}^*$. 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 have little impact on performance is that any 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

Algorithm 10: node2vec random walk

Result: Walks sampled from $\mathcal{G}_r$: $\{ \boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_b \}$
Input: $\mathcal{G}_r$: training graph ;
b: number of walks ;
k: length of random walk ;
z_1, z_2: node2vec parameters

---

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 $v = \boldsymbol{w}_i[j-1]$ **then**
        $\alpha(v) = 1/z_1$;
      if v *is a neighbor of* $\boldsymbol{w}_i[j-1]$ **then**
        $\alpha(v) = 1$;
      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

descriptive of the graphs we want to sample (Section 8.3).

NetGAN avoids overfitting by using edge prediction on a held-out validation set $\mathcal{E}_v \subset \mathcal{E}^*$. Another negative set $\tilde{\mathcal{E}}_v$ of non-edges (i.e., $\tilde{\mathcal{E}}_v \cap \mathcal{E}^* = \emptyset$) 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 $(v,u) \in \mathcal{E}_v \cup \tilde{\mathcal{E}}_v$. 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 score does not get larger after z sequential evaluations.

Algorithm 11: Edge Prediction Stopping Criterion

Result: Yes if edge prediction has not improved for z iterations and No otherwise. The frequency matrix $\tilde{A}$ with the best edge prediction seen.
Input: $\mathcal{E}_v$: positive validation edge set ;
$\tilde{\mathcal{E}}_v$: negative validation edge set ;
$\tilde{B}$: most recent model ;
best: highest edge prediction seen ;
evalsLeft: number of evaluations left before terminating ;
z: tolerance for no improvement ;
$\tilde{A}$: frequency matrix with best edge prediction seen ;

True score vector $\mathbf{x} = [1 \text{ for } (v,u) \in \mathcal{E}_v, 0 \text{ for } (v,u) \in \tilde{\mathcal{E}}_v]$;
Valid score vector $\mathbf{y} = [\tilde{b}_{v,u} \text{ for } (v,u) \in \mathcal{E}_v \cup \tilde{\mathcal{E}}_v]$;
if $AUC(\mathbf{x},\mathbf{y}) + AP(\mathbf{x},\mathbf{y}) \leq best$ **then**
  Decrement evalsLeft;
else
  evalsLeft $= z$;
  $best = \text{AUC}(\mathbf{x},\mathbf{y}) + \text{AP}(\mathbf{x},\mathbf{y})$;
  $\tilde{A} = \tilde{B}$;
if *evalsLeft is 0* **then**
  return Yes;
else
  return No;

AUC ROC and AP (Powers, 2011)

The *Receiving Operatic Characteristic* ROC curve captures the ability of threshold binary classifiers to correctly label one-dimensional data $\mathbf{y}$ using true binary labels $\mathbf{x}$. The ROC curve is $(\text{FPR}(t), \text{TPR}(t))$ where $\text{FPR}(t)/\text{TPR}(t)$ 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 is defined as 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 $t \leq t^*$, $\mathrm{FPR}(t) = 0$. For all $t \geq t^*$, $\mathrm{TPR}(t) = 1$. 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 consists of points $(\mathrm{R}(t), \mathrm{Pr}(t))$ where recall that $\mathrm{R}(t)$ is the fraction of values $x_i = 1$ that are labeled 1 by threshold t (indices i with $y_i > t$) and the precision $\mathrm{Pr}(t)$ 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: $\tilde{A}$: frequency matrix ;
m^*: number of target edges ;
$\mathcal{V}^*$: target nodes ;

Initialize $\mathcal{E} = \emptyset$;
$m' = 0$;
Permute $\mathcal{V}^*$ with uniformly random permutation π ;
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* $\mathcal{E}$ **then**
    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 $\overline{\mathcal{E}} = \mathcal{V}^* \times \mathcal{V}^* \setminus \mathcal{E}$ 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 $\tilde{a}_{v,u}$ without replacement. The main reason for sampling at least one edge for each node is to make disconnected graphs less likely. We consider a much simpler and independent graph sampling in Section 8.5.

8.2 Our contribution

We conduct a principled investigation of NetGAN and its 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).

We focus on the following questions around 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 by Grover and Leskovec (2016) with second-order memory 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) use 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 8.7.1).

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 (v, u) independently in $\mathcal{G}$ with probability essentially proportional to $\tilde{a}_{v,u}$ (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 the uniform probability $\frac{1}{2m^*}$ induced by the standard random walk. 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 (Section 5.1.2, Theorem 5.1.4). 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 S so as to maximize the second-largest eigenvalue modulus (SLEM) $\mu(S) = \max_{i=2,\ldots,n} |\lambda_i(S)| = \max\{\lambda_2(S), -\lambda_n(S)\}$ 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. This method becomes too slow for instances larger than roughly 1000 edges. For larger graphs, also following Boyd et al. (2004), we use *subgradient* methods by Overton and Womersley (1993) to approximately compute the FMMC.

Next, we derive some intuition for 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 5.1.3). 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 (for reference, see (Horn and Johnson, 1990)), we can write $\lambda_2(S)$ and $-\lambda_n(S)$ as follows:

$$\lambda_2(S) = \sup\{\mathbf{x}'S\mathbf{x} \ : \ ||\mathbf{x}||_2 \leq 1, \mathbf{1}'\mathbf{x} = 0\},$$
$$-\lambda_n(S) = \sup\{-\mathbf{x}'S\mathbf{x} \ : \ ||\mathbf{x}||_2 \leq 1\},$$

Expanding $\mathbf{x}'S\mathbf{x} = \sum_{v,u} x_v x_u s_{v,u}$ shows that for a given chain S, the entries of the eigenvector $\mathbf{x}_2$ corresponding to $\lambda_2(S)$ will tend to have opposite signs when vertices are sparsely connected, and the same sign when they are densely connected. Thus, to minimize $\lambda_2(S)$, it is good to put

large weight on entries $s_{v,u}$ connecting the sparsely connected Fiedler cut associated with $\mathbf{x}$.

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

$$\mu(\widetilde{S}) \geq \mu(S) + Tr(J(\widetilde{S} - S))$$

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

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

$$\begin{aligned}
\mu(\widetilde{S}) &\geq \lambda_2(\widetilde{S}) \\
&\geq \mathbf{x}_2'\widetilde{S}\mathbf{x}_2 \\
&= \mathbf{x}_2'\widetilde{S}\mathbf{x}_2 + \mu(S) - \mathbf{x}_2' S \mathbf{x}_2 \\
&= \mu(S) + \mathbf{x}_2'(\widetilde{S} - S)\mathbf{x}_2 \\
&= \mu(S) + \sum_{i,j} x_{2,i} x_{2,j} (\widetilde{s}_{i,j} - s_{i,j}) \\
&= \mu(S) + Tr((\mathbf{x}_2\mathbf{x}_2')(\widetilde{S} - S)).
\end{aligned}$$

A similar computation shows that when $\mu(S) = -\lambda_n$, $J = -\mathbf{x}_n'\mathbf{x}_n$ is a subgradient. The subgradient method in each step moves from S to $S - \alpha J$, for some step size α. When $S - \alpha J$ is outside the feasible set $\{S : S \geq 0, S\mathbf{1} = \mathbf{1}\}$, 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 S

Result: Transition matrix S of the Fastest-mixing Markov chain on $\mathcal{G}_r$;
Input: Graph $\mathcal{G}_r = (\mathcal{V}^*, \mathcal{E}_r)$;
numIters: number of steps ;

Initialize S 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 $s_{v,u}$ for $(v,u) \in \mathcal{E}_r$;
for $i \leftarrow 1$ **to** *numIters* **do**
  Compute sub-gradient: $\mathbf{j}^{(i)}$ of $\mathbf{p}$: $\mathbf{j}^{(i)}_{v,u} = (\mathbf{x}_v - \mathbf{x}_u)^2$;
  Take Step: $\mathbf{p} = \mathbf{p} - \alpha_i \mathbf{j}^{(i)} / \parallel \mathbf{j}^{(i)} \parallel_2$;
  Project to positive orthant: ;
  for $v \in \mathcal{V}^*$, $\mathcal{I}(v) = \{e = (v,u) \ : \ (v,u) \in \mathcal{E}_r\}$ **do**
    while $\sum_{e \in \mathcal{I}(v)} p_e > 1$ **do**
      $\delta = \min\{\min_{e \in \mathcal{I}(v)} p_e, (\sum_{e \in \mathcal{I}(v)} p_e - 1)/|\mathcal{I}(v)|\}$;
      for $e \in \mathcal{I}(v)$ **do**
        $p_e = p_e - \delta$
      end
      $\mathcal{I}(v) = \{e = (v,u) \ : \ (v,u) \in \mathcal{E}_r\}$;
    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 $p_{(v,u)}$;
    $s_{v,u} = s_{v,u} = p_{(v,u)}$;
  end
  $s_{v,v} = 1 - q_v$;
end

8.3.1 Combining the Standard random walk and Fastest-mixing Markov 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 $s_{v,u} = 0$ 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 S, and the transition matrix of the standard random walk by Q. We consider the walk with transition matrix $\frac{1}{2}S + \frac{1}{2}Q$ 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 $\frac{1}{2}S + \frac{1}{2}Q$ 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 $\frac{1}{2}S + \frac{1}{2}Q$ 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 S, 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 S, and (3) the combination walk $\frac{1}{2}S + \frac{1}{2}Q$, we compute, for each chain, the stationary distribution using edges as states. While the stationary distributions for S 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 the first vertex on edge f denoted

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 S and $\frac{1}{2}S + \frac{1}{2}Q$ 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.

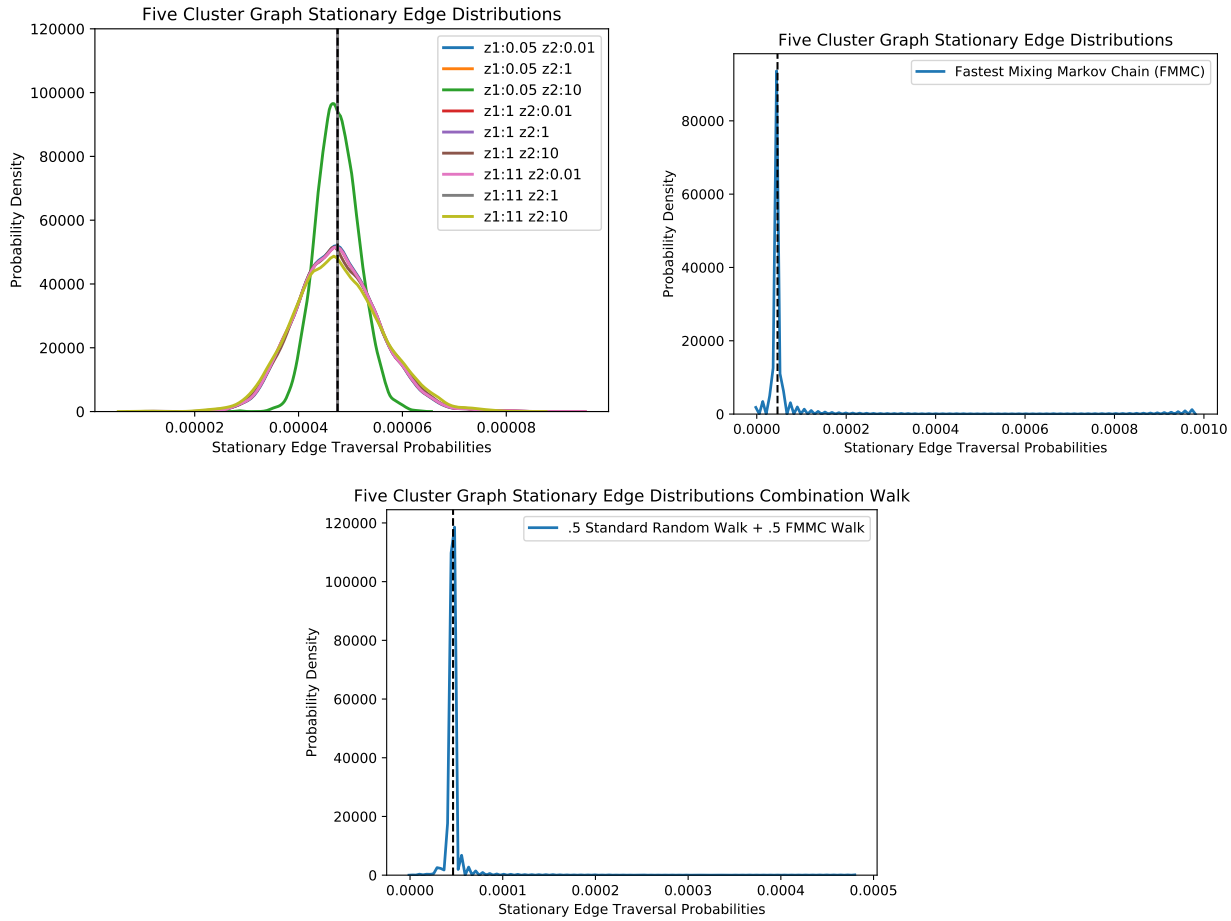


Figure 8.1: Stationary distributions over directed edges for the node2vec walk, Combination walk, and Fastest-mixing Markov chain.

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 8.7.2. 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 8.7.2, 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)

Algorithm 14: ScaleMatrix

Result: A: probabilistic adjacency matrix
Input: $\tilde{A}$: frequency matrix ;
m^*: number of edges ;

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 $X = \emptyset$ **then**
    break ;
  else
    $Z = \sum_{(v,u)\in X} a_{v,u}$;
    $a_{v,u} = \min(1, \delta \cdot a_{v,u}/Z)$ for all $(v,u) \in X$;
    $\delta = m^* - \sum_{v,u} a_{v,u}$;
end

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 $X = \{(v,u) \ : \ a_{v,u} \in (0,1)\}$ is non-empty, we let $Z = \sum_{(v,u)\in X} a_{v,u}$ and $a_{v,u} = \min(1, \delta \cdot (a_{v,u}/Z))$. 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

Bojchevski et al. (2018) use ROC AUC and AP to curtail NetGAN training before it overfits. As noted above in Section 8.1.4, if there is a perfect threshold classifier then edge classification is perfect, as 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 have 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 additionally uses the expected spectral gap of the generated graph measured by the spectrum of $L(A)$. As discussed in Section 5.1.1, Chung and Radcliffe (2011) show that when a graph $\mathcal{G}$ with adjacency matrix B is generated using independent edge sampling on A, the spectral gap of $L(B)$ is sharply concentrated around that of $L(A)$. Hence, the spectral gap of $L(A)$ is a good approximation for that of the graph generated from A using independent edge sampling. 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 8.7.1 and 8.7.2 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 similarity of graphs and the diversity metrics (Chapter 3) using abbreviations and symbols. Table 8.2 is a reference to these abbreviations.

8.7.1 What is the NetGAN learning?

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 R at stationarity is equal to $\pi_v r_{v,u}$. The distribution $z^{(R)}$ is the distribution over edges induced by R at stationarity.

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 their scores did not improve. Otherwise, No.

Input: $\mathcal{E}_v$: positive validation edges ;
$\tilde{\mathcal{E}}_v$: negative validation edges ;
$\tilde{B}$: current frequency matrix ;
$\tilde{A}$: most recent frequency matrix that had the best AP, AUC, or SpecGap ;
$\lambda_2(A^*)$: spectral gap of input matrix ;
bestAUC, bestAP, bestSpecGapDiff: Best scores found for each evaluation criterion ;
evalsLeftAUC, evalsLeftAP, evalsLeftSpecGap: Number of evaluations remaining for each evaluation criterion before termination ;
z: Number of evaluations tolerated without improving ;

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 $\mathbf{y} = [\tilde{b}_{u,v} \text{ for } (v,u) \in \mathcal{E}_v \cup \tilde{\mathcal{E}}_v]$;
Let $B = \text{ScaleMatrix}(\tilde{B})$ (Algorithm 14) ;
if *evalsLeftAUC* > 0 **then**
  if $AUC(\mathbf{x},\mathbf{y}) \leq bestAUC$ **then**
    Decrement evalsLeftAUC
  else
    bestAUC $= \text{AUC}(\mathbf{x},\mathbf{y})$;
    evalsLeftAUC $= z$;
    $\tilde{A} = \tilde{B}$;
if *evalsLeftAP* > 0 **then**
  if $AP(\mathbf{x},\mathbf{y}) \leq bestAP$ **then**
    Decrement evalsLeftAP
  else
    bestAP $= \text{AP}(\mathbf{x},\mathbf{y})$;
    evalsLeftAP $= z$;
    $\tilde{A} = \tilde{B}$;
if *evalsLeftSpecGap* > 0 **then**
  if $|\lambda_2(B) - \lambda_2(A^*)| \geq bestSpecGapDiff$ **then**
    evalsLeftSpecGap $=$ evalsLeftSpecGap $- 1$
  else
    bestSpecGapDiff $= |\lambda_2(A') - \lambda_2(A^*)|$;
    evalsSpecGap $= z$;
    $\tilde{A} = \tilde{B}$;
if *evalsLeftAP* $=$ *evalsLeftSpecGap* $=$ *evalsLeftAUC* $= 0$ **then**
  Return Yes
else
  Return No
end

Full name	Abbreviated name
Edge overlap: Fraction of probability mass in a probabilistic adjacency matrix that is on $\mathcal{E}^*$	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.

2. The distribution $y^{(\tilde{A})}$ with $y^{(\tilde{A})}(v,u) = \tilde{a}_{v,u}/Z$ where $Z = \sum \tilde{a}_{v,u}$.

We track[6] $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(R)})$ as a proxy for how close the edge distribution learned by NetGAN is to the edge distribution induced by walk transition matrix R.

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 (v,u) 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" $(v,u) \notin \mathcal{E}^*$. We call mass on "noise pairs" addition noise (in contrast to subtraction noise which would be omitting traversals over pairs $(v,u) \in \mathcal{E}^*$). We compute the average $\overline{x} = (\sum_{(v,u)\notin\mathcal{E}^*} a_{v,u})/((n^*)^2 - |\mathcal{E}^*|)$ to measure the average addition noise. For both walks, once $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(R)}) \leq .6$, we observe that the average deviation $|a_{v,u} - \overline{x}|$ is at least $\overline{x}$; for the NETSCIENCE and AIRPORT graphs, this holds as early as $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(R)}) \leq .9$ (Figure 8.2). We also see that for both walks, the average addition noise decreases with $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(R)})$, indicating that the GAN learns to add probability mass on pairs in $\mathcal{E}^*$ instead of addition noise.

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 2.2.1) than there would be if noise were

[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.

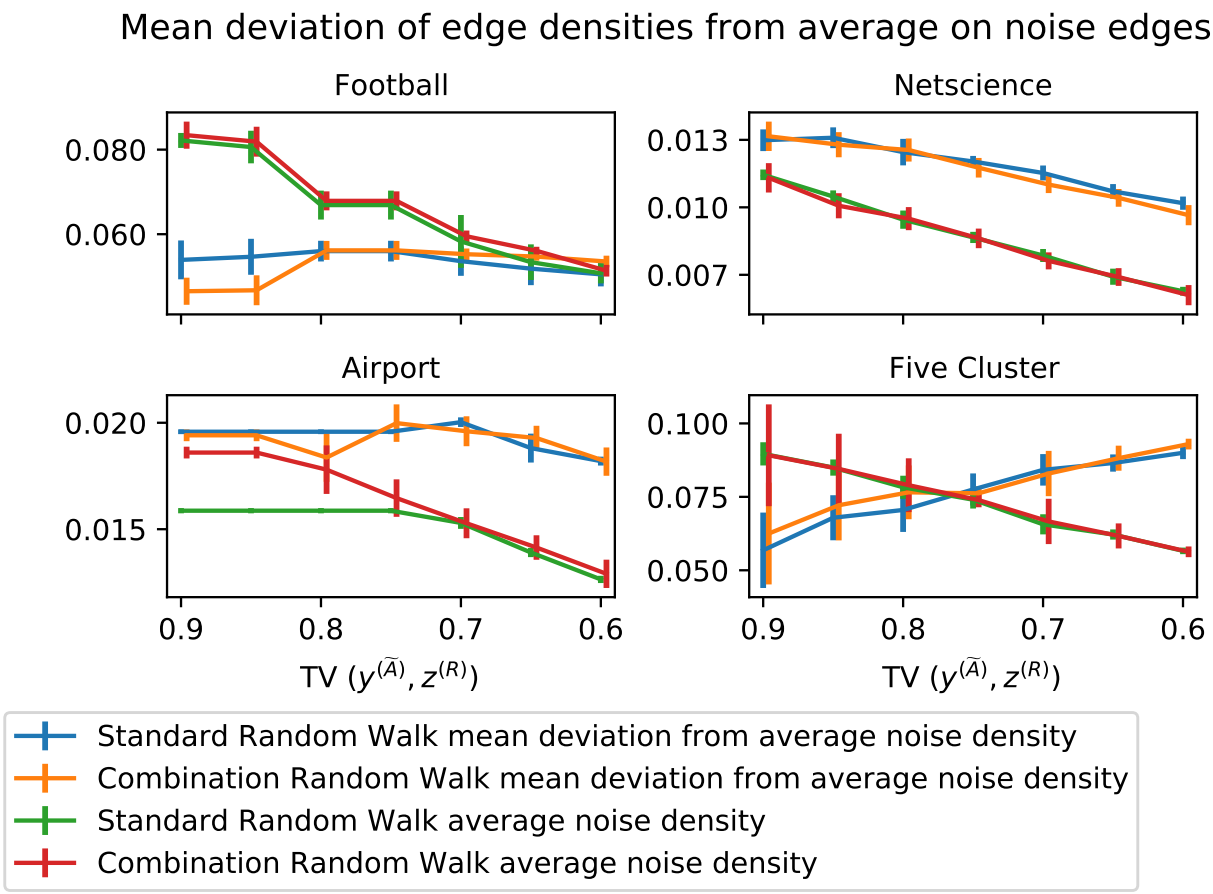


Figure 8.2: Average deviation of addition noise from uniform against the Total variation distance between the edge densities learned by the NetGAN and the random walk edge traversal probabilities. The x-axis is labeled in decreasing Total variation distance so that the number of training iterations is increasing from left to right.

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.

8.7.2 Comparing NetGAN Variants

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 for both stopping methods that the standard walk generally outperforms the combination walk on matching the spectrum using the ℓ_2^{LW} metric,

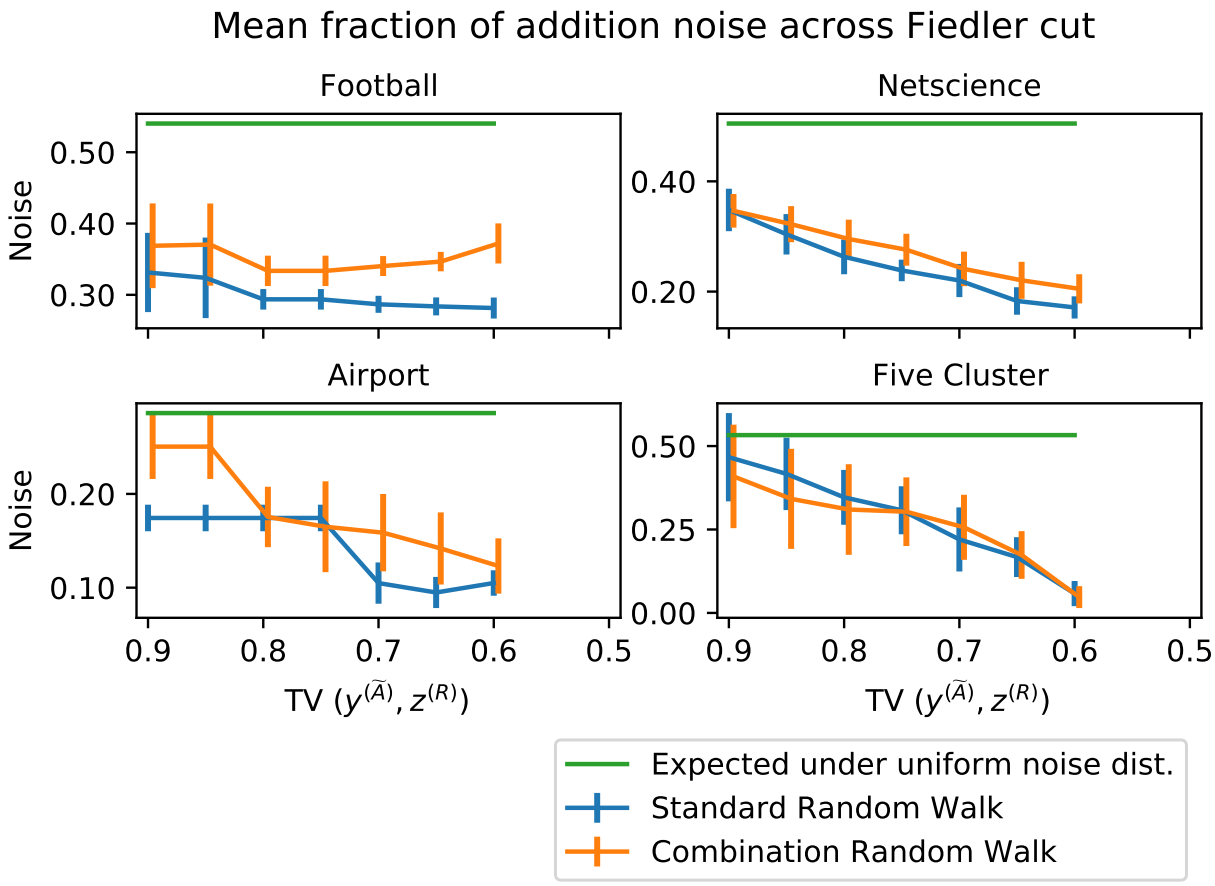


Figure 8.3: Addition noise across Fiedler cut against the Total variation distance between the edge densities learned by the NetGAN and the random walk edge traversal probabilities. The x-axis is labeled in decreasing Total variation distance so that the number of training iterations is increasing from left to right.

but the results are mixed. Any gains in matching the spectrum typically come with a decrease in entropy. Most consistently, we notice that for both stopping criteria $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(R)})$ is smaller for the standard walk than the combination walk.

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

Walk/Stop	$d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(R)})$	$\lvert\lambda_2^* - \lambda_2\rvert$	ℓ_2^{LW} Spec.	ROC AUC	AP	EO	H(A)
Football							
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 $d_{\text{TV}}(y^{(\tilde{A})}, z^{(R)})$ in Figure 8.4. Controlling for $d_{\text{TV}}(y^{(\tilde{A})}, z^{(R)})$, we observe that the combination walk at least matches or has smaller/larger ℓ_2^{LW}/entropy. These experiments suggest that if NetGAN is trained with the combination walk until it reaches a $d_{\text{TV}}(y^{(\tilde{A})}, z^{(R)})$ threshold, ℓ_2^{LW} will be smaller and entropy larger than if trained with the standard walk. For ROC AUC, AP, and EO the results vary depending on which walk has the larger value.

Typically the SP+EP stopping criterion was 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, $d_{\text{TV}}(y^{(\tilde{A})}, z^{(R)})$, 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.

Discrete graph sampling and matching discrete graph properties

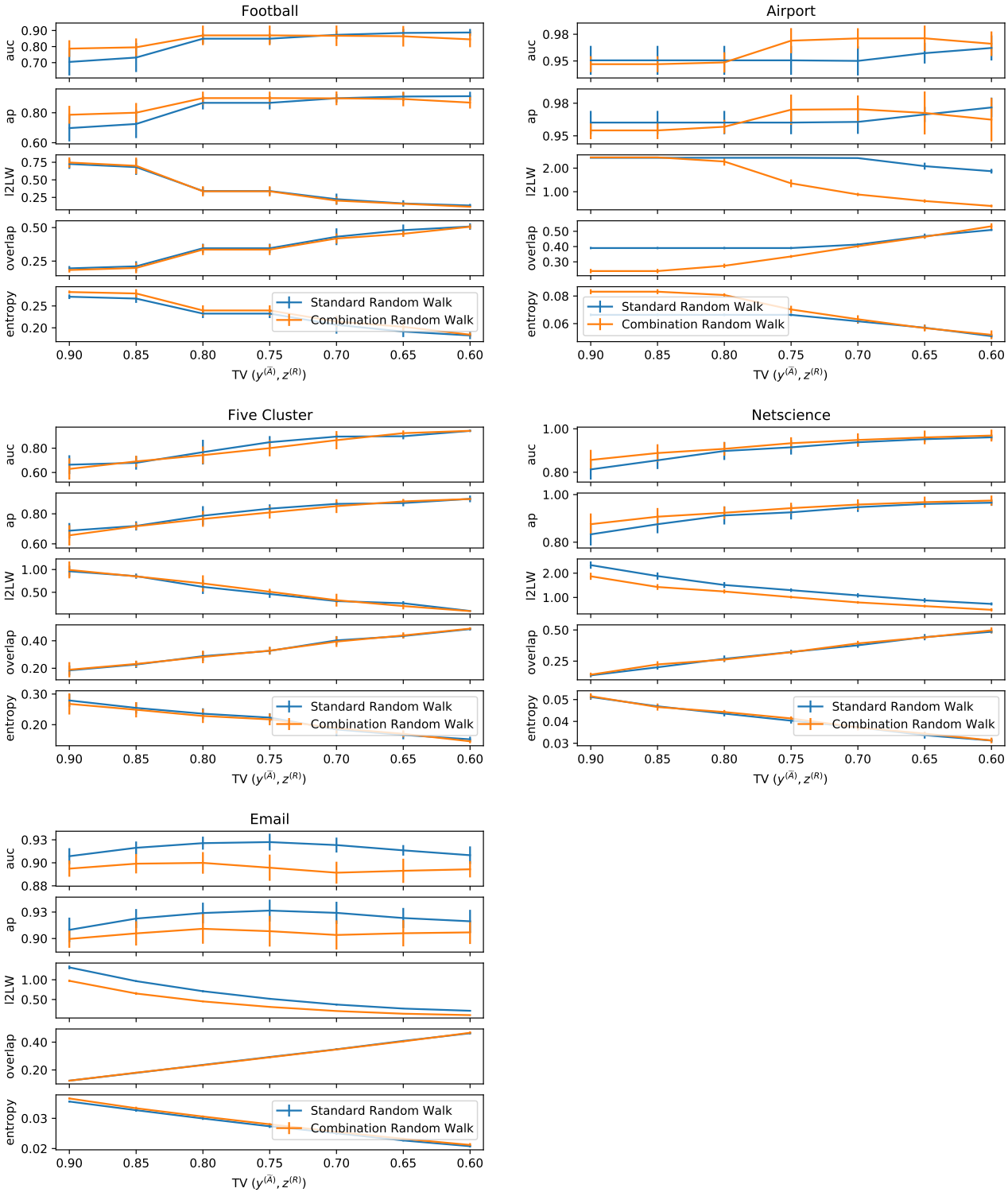


Figure 8.4: Statistics of probabilistic adjacency matrix A against the Total variation distance between the edge densities learned by NetGAN and the random walk edge traversal probabilities.

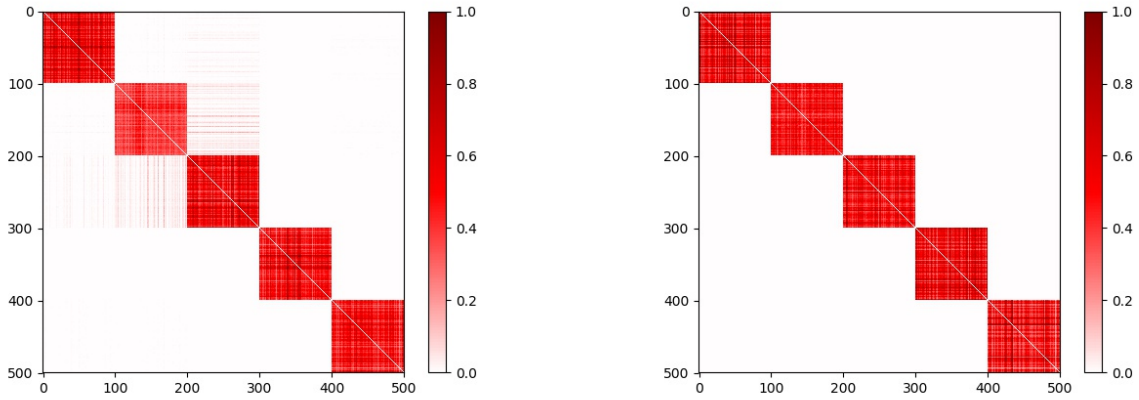


Figure 8.5: Heat maps for edge densities learned by NetGAN on the Five Cluster graph after 4,500 training iterations and 6,500 training iterations. Longer training improves the estimates of density of inter-cluster edges in the Five Cluster graph.

Table 8.4: Properties of the 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±0.01**	**0.01±0.0**	115±0	0.04±0.01	**0.06±0.01**	0.0±0.0	1.66±0.19
Combo.	0.04±0.01	0.01±0.0	**115±0**	0.05±0.01	0.08±0.02	**0.0±0.0**	**1.06±0.07**
			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±0.01	**367±9**	1.26±0.17	0.25±0.02	**0.01±0.0**	**0.27±0.08**
Combo.	**0.01±0.0**	**0.01±0.0**	362±9	**1.05±0.3**	**0.23±0.04**	0.01±0.0	0.49±0.12
			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±0.01	0.11±0.14	428±83	**2.02±0.35**	0.05±0.01	**0.01±0.0**	5.94±0.55
Combo.	**0.0±0.0**	**0.01±0.01**	**497±7**	2.5±0.37	**0.04±0.01**	0.01±0.0	**5.47±0.56**
			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±0.05**	**0.09±0.05**	476±18	0.18±0.07	**0.17±0.04**	**0.0±0.0**	**1.15±0.38**
Combo.	0.08±0.05	0.1±0.06	**479±11**	**0.12±0.04**	0.34±0.09	0.0±0.0	4.2±1.05
			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±0.02**	0.02±0.02	1068±26	0.2±0.07	**0.06±0.05**	**0.0±0.0**	**0.65±0.24**
Combo.	0.04±0.02	**0.02±0.01**	**1117±6**	**0.13±0.05**	0.11±0.04	0.0±0.0	1.84±0.18

Table 8.5: Properties of the 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.	**0.03±0.02**	**0.01±0.01**	115±0	**0.06±0.02**	**0.09±0.04**	0.0±0.0	1.84±0.14
Combo.	0.12±0.02	0.06±0.02	**115±0**	0.15±0.02	0.2±0.03	**0.0±0.0**	**1.83±0.11**
			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±0.01	0.04±0.03	**355±9**	1.48±0.25	0.32±0.08	**0.01±0.0**	**0.44±0.17**
Combo.	**0.02±0.01**	**0.02±0.01**	354±10	**1.23±0.31**	**0.3±0.08**	0.01±0.0	0.65±0.15
			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±0.01**	0.03±0.05	483±33	**2.53±0.42**	**0.05±0.02**	**0.01±0.0**	**5.28±1.44**
Combo.	0.01±0.02	**0.01±0.02**	**497±8**	2.83±0.37	0.07±0.04	0.01±0.0	5.68±1.27
			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±0.05**	**0.13±0.06**	462±15	**0.25±0.07**	**0.21±0.04**	**0.0±0.0**	**1.36±0.32**
Combo.	0.13±0.05	0.16±0.05	**482±5**	0.15±0.02	0.41±0.04	0.0±0.0	4.73±0.36
			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±0.01**	0.04±0.01	1045±12	0.26±0.04	**0.12±0.02**	**0.0±0.0**	**0.82±0.11**
Combo.	0.06±0.01	**0.03±0.01**	**1112±3**	**0.19±0.04**	0.14±0.02	0.0±0.0	1.82±0.09

Table 8.6: Properties of the 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±0.02	0.02±0.01	115±0	0.08±0.03	0.11±0.03	0.0±0.0	1.72±0.19
Combo.	**0.04±0.01**	**0.01±0.0**	**115±0**	**0.05±0.02**	**0.09±0.02**	**0.0±0.0**	**1.1±0.08**
			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±0.14	0.33±0.03	0.01±0.0	0.23±0.05
Combo.	**0.02±0.01**	**0.01±0.0**	375±2	**1.15±0.31**	**0.32±0.06**	**0.01±0.0**	**0.44±0.07**
			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±0.01	0.01±0.0	3.35±0.31
Combo.	0.0±0.0	**0.0±0.01**	**498±6**	2.62±0.34	**0.02±0.0**	0.01±0.0	**3.09±0.37**
			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±0.05**	**0.09±0.05**	**500±1**	0.18±0.06	**0.24±0.04**	**0.0±0.0**	**1.56±0.27**
Combo.	0.09±0.05	0.13±0.07	499±1	**0.14±0.04**	0.4±0.09	0.0±0.0	4.81±1.06
			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±0.02**	**0.02±0.01**	1132±4	0.17±0.04	**0.1±0.04**	**0.0±0.0**	**0.56±0.13**
Combo.	0.06±0.02	0.03±0.01	**1133±0**	**0.15±0.03**	0.14±0.03	0.0±0.0	2.2±0.19

Table 8.7: Properties of the 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±0.02**	**0.03±0.01**	115±0	**0.11±0.02**	**0.15±0.03**	0.0±0.0	**1.82±0.09**
Combo.	0.14±0.02	0.07±0.01	115±0	0.18±0.01	0.23±0.02	**0.0±0.0**	1.77±0.06
			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±0.02	0.03±0.02	**378±0**	1.44±0.19	0.43±0.08	**0.01±0.0**	**0.3±0.05**
Combo.	**0.03±0.01**	**0.02±0.01**	377±2	**1.29±0.23**	**0.42±0.08**	0.01±0.0	0.54±0.07
			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±0.01**	0.03±0.04	486±29	**2.63±0.43**	**0.04±0.03**	0.01±0.0	**3.11±1.03**
Combo.	0.02±0.02	**0.02±0.02**	**499±4**	2.92±0.36	0.06±0.05	0.01±0.0	3.46±0.93
			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±0.05**	**0.15±0.06**	**500±0**	0.25±0.05	**0.29±0.03**	**0.0±0.0**	**1.79±0.15**
Combo.	0.15±0.06	0.2±0.05	500±0	**0.17±0.02**	0.47±0.03	0.0±0.0	5.49±0.26
			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±0.02**	**0.04±0.01**	1132±4	0.2±0.02	**0.15±0.01**	**0.0±0.0**	**0.59±0.13**
Combo.	0.08±0.01	0.04±0.01	**1133±0**	**0.19±0.03**	0.16±0.01	0.0±0.0	2.2±0.08

Tables 8.4, 8.5, 8.6 and 8.7 show statistics for the largest connected component of the discrete graphs generated. Training with the 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 keeping more nodes connected but in general lagging slightly behind in all other metrics.

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. Contrary to Bojchevski et al. (2018), we normalize for the number of edges seen by the generator across walk lengths and we find that this 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 using 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.

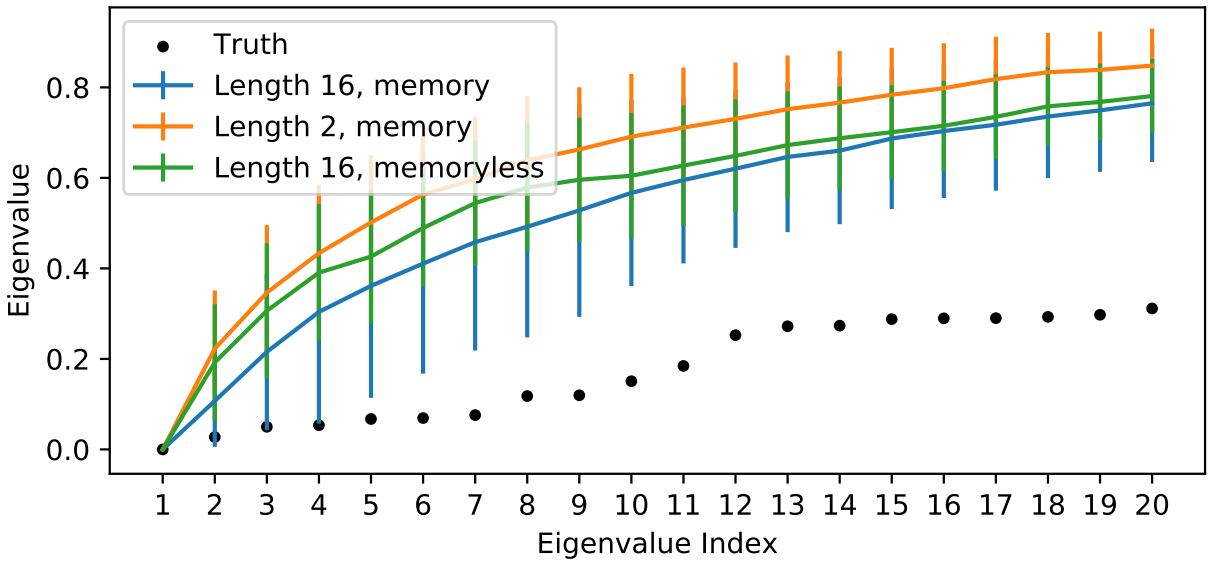


Figure 8.6: Average eigenvalues of $L(A)$ when training using the airport graph and (1) length-16 walks with memory states (2) length-2 walks with memory states and (3) length-16 walks without memory states. The average AUCs are (1) $.975 \pm .008$ (2), $.968 \pm .005$, (3).$972 \pm .006$ and average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

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 is 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

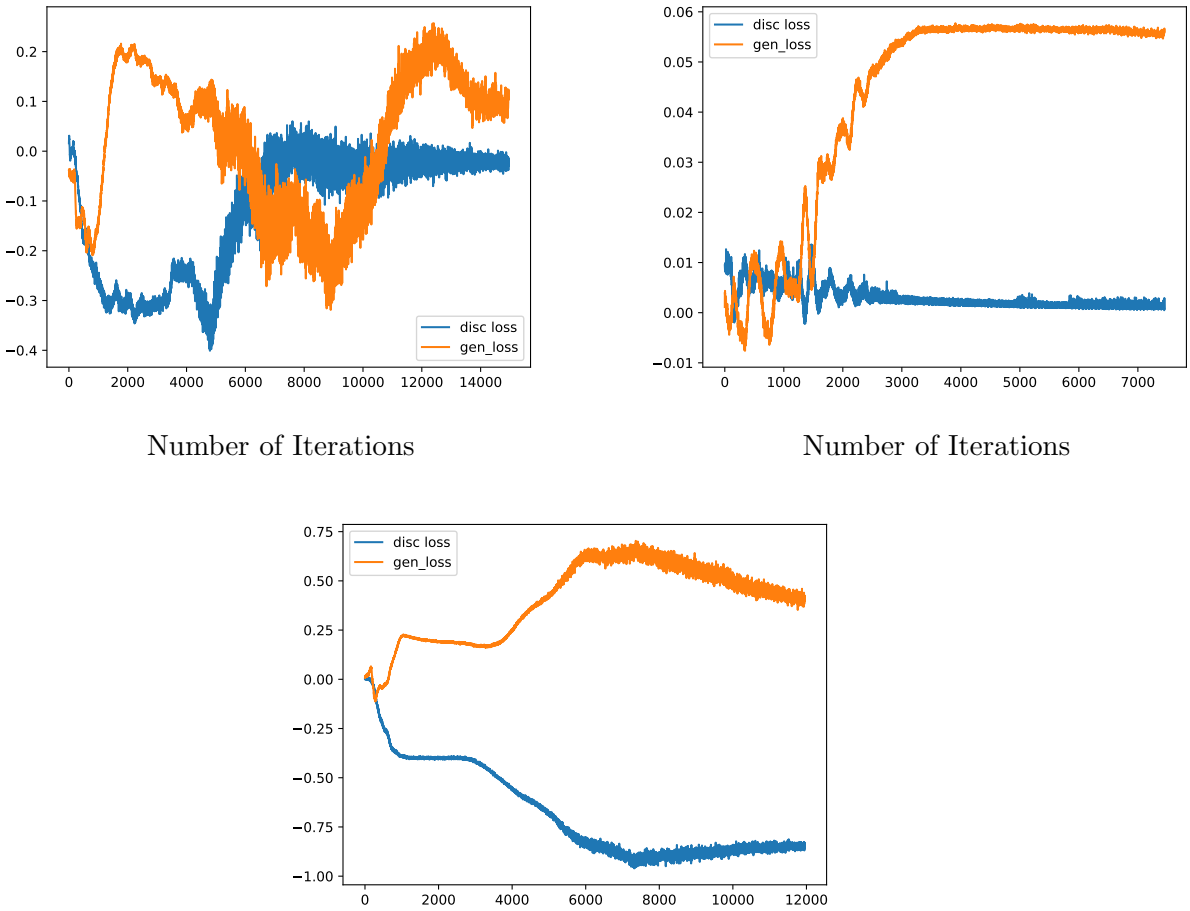


Figure 8.7: Discriminator and generator losses during training using the Five Cluster graph and (1) length-16 walks with memory states, (2) length-2 walks with memory states, and (3) length-16 walks without memory states.

120/90 configuration has lower ROC AUC/AP for $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(R)}) \leq .6$ (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 $d_{\mathrm{TV}}(y^{(\tilde{A})}, z^{(R)})$ is small.

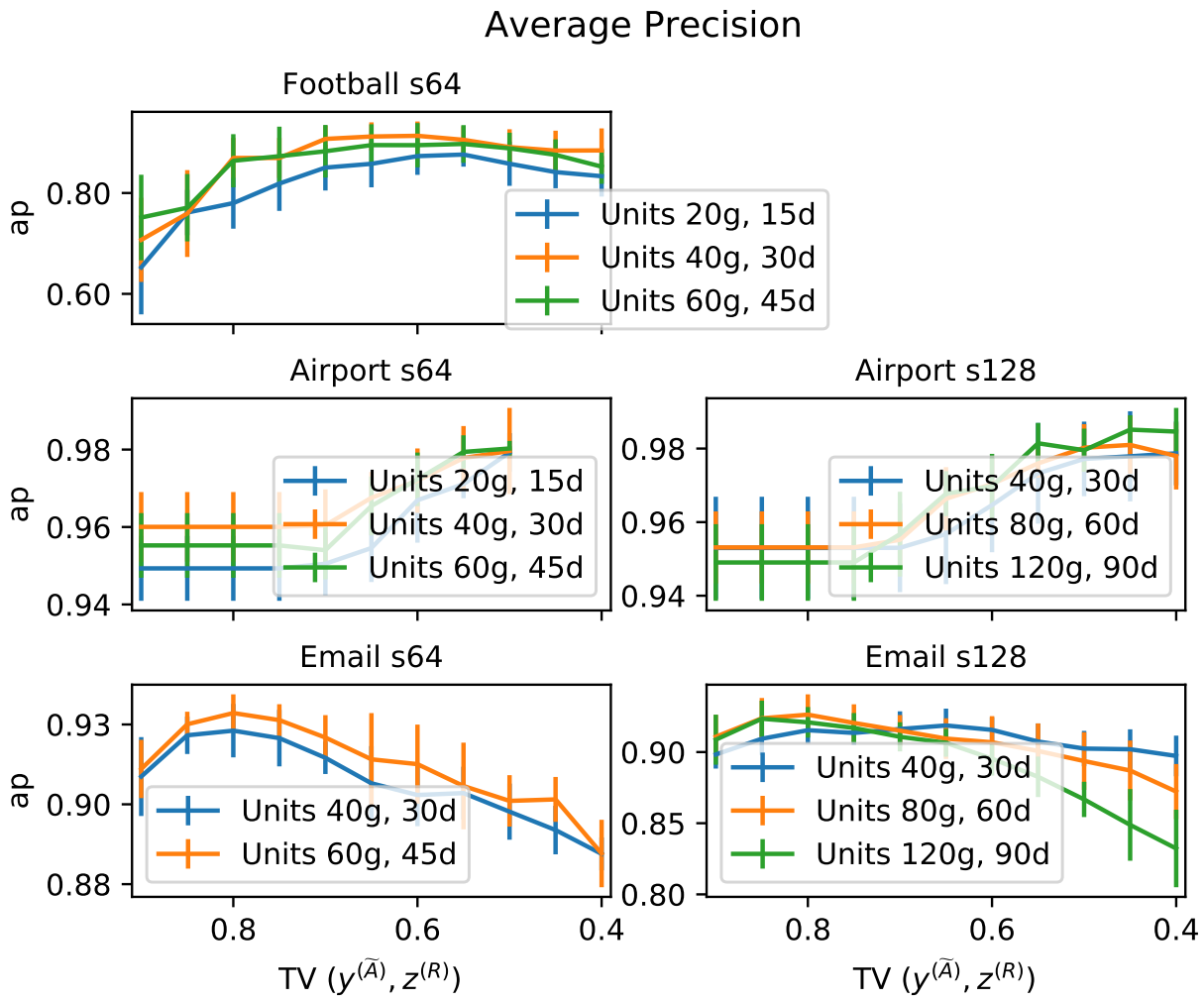


Figure 8.8: Average Precision of frequency matrices learned with different NetGAN sizes.

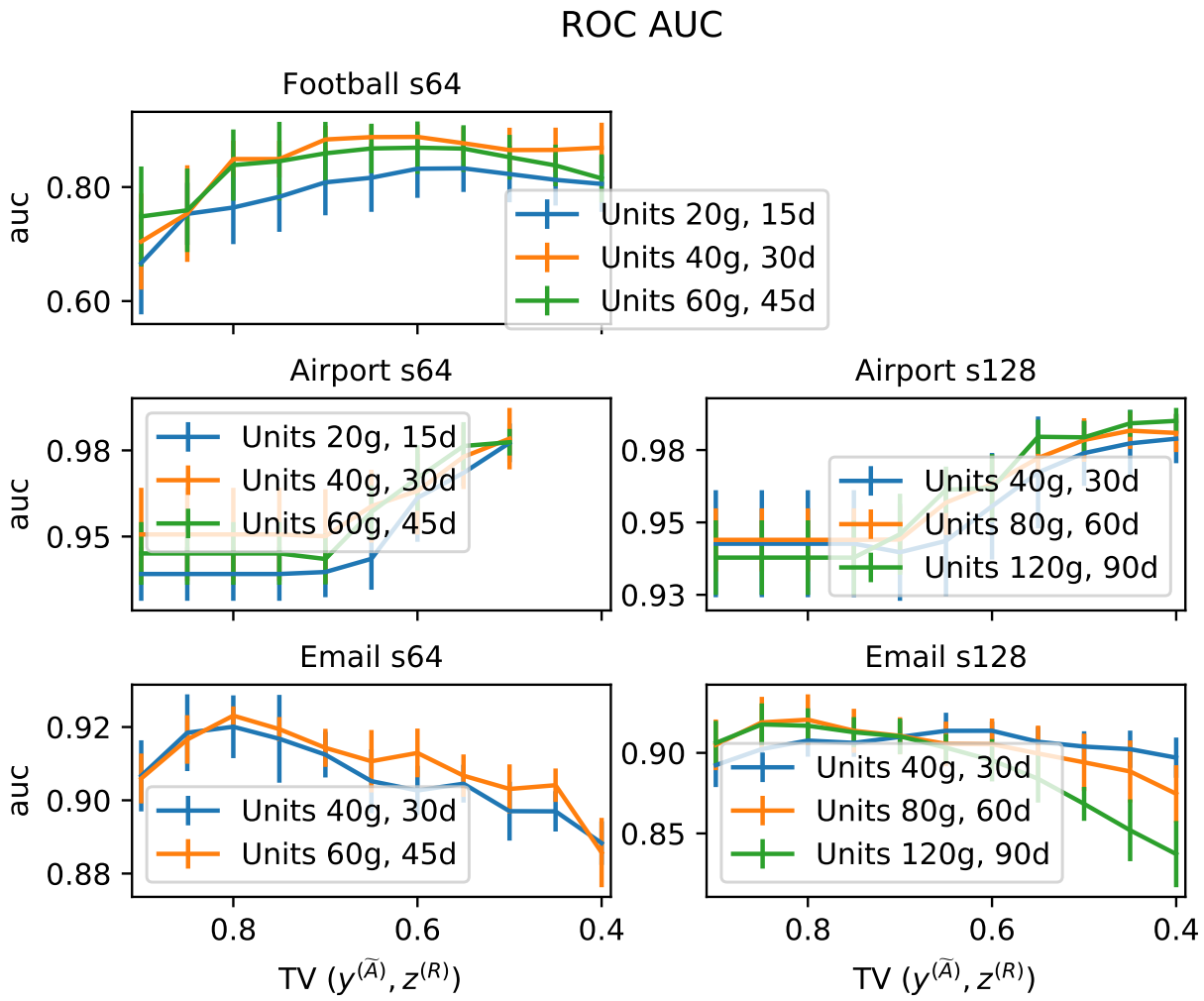


Figure 8.9: ROC AUC for frequency matrices learned with different NetGAN sizes.

Chapter 9

Random walk-based generation

9.1 Motivation

One of the shortcomings of NetGAN graph generation (Chapter 8) is how opaque it is. NetGAN trains a GAN to generate synthetic walks and uses the frequencies with which edges appear on those walks to build a probabilistic adjacency matrix which is used as a graph generator using independent edge sampling. 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 adding variation among the output graphs. Recall from the discussion in Section 8.7.1 that *addition noise* adds probability mass to pairs of vertices that do not appear in $\mathcal{E}^*$ (as opposed to removal noise, which removes 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. In this chapter, we design a technique that uses 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 8.7.1). Is adding noise on either side of sparse cuts within dense parts 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 without training a GAN?

9.2 Overview

Our *Random walk-based generation* approach constructs a probabilistic adjacency matrix[1] A by sampling collections of walks $\mathcal{W}$. All walks in Random walk-based generation are standard random walks[2]. The guiding principle of generating A is that if two nodes v and u are on opposite sides of a low-conductance cut $S \subseteq \mathcal{V}^*$ (meaning that $v \in \mathcal{S}$, $u \in \overline{\mathcal{S}}$), then two random walks starting at v and u will with high probability take many steps before they meet (Proposition 5.1.1). Suppose we generate many random walks starting from node v and for all $v \neq u$ we assign weights $a_{v,u}$ that increase with the number of times node pairs v and u appeared on the same walk and decrease with the number of steps between them on those walks. A walk of length k leaves a cut $\mathcal{S}$ if $\boldsymbol{w}[1] = v \in \mathcal{S}$ and there exists some $t \leq k$ such that $\boldsymbol{w}[t] = u \in \overline{\mathcal{S}}$. Then, by the property that random walks take many steps before leaving low-conductance cuts, nodes u that appear in highly clustered parts of $\mathcal{G}^*$ with v should have high $a_{v,u}$ and those u separated from v by sparse cuts should have small $a_{v,u}$. So if each edge (v, u) is included in $\mathcal{G}$ using independent edge sampling 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). The 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, the frequency matrix $\tilde{A}$ is constructed in rounds that each have two phases.

1. In the *cut construction* phase described in Section 9.4, b walk lists $\mathcal{WS} = \mathcal{W}_1, \mathcal{W}_2 \ldots \mathcal{W}_b$ are generated. To sample $\mathcal{WS}$, first two disjoint subsets of nodes $\mathcal{S}_1, \mathcal{S}_2$ are sampled using two independent random walks. Then, more walks are sampled to complete each $\mathcal{W}_i \in \mathcal{WS}$ and the collection of walks $\mathcal{WS}$ define a random cut $\mathcal{S}$ such that $\mathcal{S}_1 \subseteq \mathcal{S}$ and $\mathcal{S}_2 \subseteq \overline{\mathcal{S}}$.

2. In the *frequency matrix update* phase, $\tilde{A}$ is updated for each $\boldsymbol{w} \in \mathcal{W}_i \in \mathcal{WS}$. Initially, each

[1] Probabilistic adjacency matrix is defined in Section 2.2.
[2] Random walk definitions are in Section 2.4.

entry $\tilde{a}_{v,u}$ is zero. Entries $\tilde{a}_{v,u}$ are increased based on how many times pairs (v, u) appear on walks $\boldsymbol{w} \in \mathcal{W}_i$, how many steps v and u are apart on the walks, and whether or not they are both in $\mathcal{S}$ or $\overline{\mathcal{S}}$ (Section 9.5).

After a sufficient number of rounds has been 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 adjacency matrix A

Input: $\mathcal{G}^*$: input graph ;

b: number of walk iterations per batch ;

k: minimum number of nodes needed to be seen by initialization walks ;

zeroCross: flag that says whether or not to add mass across cuts ;

walkDistWeight: flag that says whether or not to weight the amount of mass added using the number of steps between nodes ;

T: total number of walk algorithm iterations ;

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) ;

  $\tilde{A} =$ matrixUpdate$(\tilde{A}, \mathcal{S}, \mathcal{WS}, \text{zeroCross}, \text{walkDistWeight})$ (Algorithm 19);

end

Construct probababilistic adjacency matrix $A =$ scaleMat$(\tilde{A}, m^*)$ (Algorithm 14) ;

9.3 Related work

Our problem of assigning probability $a_{v,u}$ to include (v, u) 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 exists 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 are not new. NIBBLE by Spielman and Teng (2004) 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 the cut; this relationship is one of the key properties behind our approach of using random walks to discover sparse cuts. Spielman and Teng (2004) use the "lazy" random walk that uses the standard random walk with probability $\frac{1}{2}$ and stays at the current node with probability $\frac{1}{2}$. Recall from Proposition 5.1.1 that 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 $1 - t\varphi_{\mathcal{G}^*}(\mathcal{S})/2$. Following Spielman and Teng (2004), 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. Random walks are no longer needed because the page-rank vector encodes information about the *hitting times* of node pairs and hitting times are enough to discover low-conductance partitions. 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 $\mathcal{G}^*$ using projections of the *Laplacian* matrix[3]. The coarse graph has smaller size than the target. A coarsening is a membership mapping of each node in the target graph to a node in the coarse graph. The projection of the Laplacian describes the strengths of the edges connecting the nodes in the coarse graph. A random process is performed to project the graph back up to the original size. This process uses the Laplacian projection so edges only exist between node pairs that are mapped to connected nodes in the coarse graph. Further randomized edge additions and deletions are performed that are designed to preserve the node pair shortest path distribution. This is different from our method because it finds dense regions by using projections of the Laplacian instead of random walks to match the density of regions in the output graphs.

[3]The Laplacian matrix $L = D - A$ is the difference of the diagonal degree matrix D and adjacency matrix A defined in Section 2.1.

9.4 Cut Construction stage

In each round, the algorithm constructs a cut $\mathcal{S} \subseteq \mathcal{V}^*$ with a batch of walk lists $\mathcal{WS}$. The i-th walk list in $\mathcal{WS}$ is denoted by $\mathcal{W}_i$ 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*. The seed walks are started from two different vertices and run until one hits a vertex the other has hit in previous steps or they intersect at some step $t+1$. The walks are now terminated and disjoint node sets $\mathcal{S}_1$ and $\mathcal{S}_2$ are the first t nodes hit by $\boldsymbol{q}[1]$ and $\boldsymbol{q}[2]$ respectively. The sampling of the seed walks is described in Algorithm 17.

Algorithm 17: SampleSeedWalks

Result: $\boldsymbol{q}[1]$, $\boldsymbol{q}[2]$, $\mathcal{S}_1$, $\mathcal{S}_2$;
Input: $\mathcal{G}^*$: input graph ;
k: minimum number of nodes needed to be seen by initialization walks ;

Initialize seed node sets to empty, $\mathcal{S}_1 = \emptyset$ and $\mathcal{S}_2 = \emptyset$;
while $|\mathcal{S}_1| < k$ *or* $|\mathcal{S}_2| < k$ **do**
  Choose random nodes $u \neq v$ with probability proportional to d_u, d_v ;
  Initialize seed walks $\boldsymbol{q}[1](1) = u$, $\boldsymbol{q}[2](1) = v$;
  $t = 1$;
  while $\boldsymbol{q}[1](t) \neq \boldsymbol{q}[2](t)$ **do**
    if $\boldsymbol{q}[1](t) \in \mathcal{S}_2$ **then**
      Remove $\boldsymbol{q}[1](t)$ from $\boldsymbol{q}[1]$ and $\boldsymbol{q}[2](t)$ from $\boldsymbol{q}[2]$;
      break.
    if $\boldsymbol{q}[2](t) \in \mathcal{S}_1$ **then**
      Remove $\boldsymbol{q}[1](t)$ from $\boldsymbol{q}[1]$ and $\boldsymbol{q}[2](t)$ from $\boldsymbol{q}[2]$;
      break.
    Add $\boldsymbol{q}[1](t)$ to $\mathcal{S}_1$;
    Add $\boldsymbol{q}[2](t)$ to $\mathcal{S}_2$;
    $t = t + 1$;
    $\boldsymbol{q}[1](t)$ and $\boldsymbol{q}[2](t)$ sampled using the standard random walk (Section 2.4) ;

After the seed walks are terminated, walks are added to $\mathcal{W}_i$ until all nodes have been seen at least once or a maximum number of walks has been added. Once a node is seen by a walk it is marked and placed in marked node set V'. Initially, only nodes in $\mathcal{S}_1$ and $\mathcal{S}_2$ are marked. While sampling random walks, a voting map g is maintained that maps each node in $\mathcal{V}^*$ to an integer. The map g initially maps each node in $\mathcal{S}_1$ to 1, each node in $\mathcal{S}_2$ to -1 and all others to 0. The random walks are initialized from nodes not in V' and run until they hit a node in $\mathcal{S}_1$ or $\mathcal{S}_2$. If the walk hits a node $u \in \mathcal{S}_1$, then $g(v)$ is incremented for all nodes v traversed on the walk (as many times as it appears on the walk). Alternatively, if the walk hits a node $u \in \mathcal{S}_2$ first, then $g(v)$ is

decremented for all nodes v traversed by the walk. Once the walk hits a node in $\mathcal{S}_1$ or $\mathcal{S}_2$, the walk is terminated and added to $\mathcal{W}_i$. Every node on the walk is added to the marked node set V'. Walks are sampled starting from unmarked nodes not in V' until all nodes are marked and $V' = \mathcal{V}^*$ or the algorithm reaches a maximum number of walks ℓ; this completes the construction of walk list $\mathcal{W}_i$. Walk lists are built in this manner until all b walk lists are finished at which point the nodes are divided according to g. If $g(v) > 0$, then v was on more walks that hit $\mathcal{S}_1$ than those that hit $\mathcal{S}_2$. All nodes with $g(v) > 0$, including nodes in $\mathcal{S}_1$, are placed in $\mathcal{S}$. The full cut construction algorithm is described in Algorithm 18.

Our idea behind stopping walks once they hit one of the labeled nodes is to discover sparse cuts. Suppose that there exists a sparse cut $\mathcal{S} \subseteq \mathcal{V}^*$ such that $\mathcal{S}_1 \subseteq \mathcal{S}$ and $\mathcal{S}_2 \subseteq \overline{\mathcal{S}}$. Due to the sparsity of $\mathcal{S}$, we would expect most walks starting in $\mathcal{S} \setminus \mathcal{S}_1$ to hit $\mathcal{S}_1$ before $\mathcal{S}_2$. The idea is that if there are enough walks in $\mathcal{WS}$ then we should be able to approximately find $\mathcal{S}$ if $\mathcal{S}$ is sufficiently sparse.

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

9.5 Constructing a frequency matrix from walk and cut collections

After constructing $\mathcal{S}$ and $\mathcal{WS}$, the algorithm enters the *frequency matrix update* phase of the round and adds positive weight to entries of $\tilde{A}$. For each walk $\boldsymbol{w} \in \mathcal{W}_i \in \mathcal{WS}$, the matrix $\tilde{A}$ is updated by adding positive q to $\tilde{a}_{v,u}$ for pairs $(v, u) = (\boldsymbol{w}(t), \boldsymbol{w}(t+z))$ where $\boldsymbol{w}(t)$ is the node hit at time t by $\boldsymbol{w}$. The amount q added to $\tilde{a}_{v,u}$ can be a function of z so that q is larger for pairs of nodes that appear closer in $\boldsymbol{w}$ (*walk distance weight*). Another variant we explore is non-zero q only for $t \leq \ell'$ where ℓ' is the step where $\boldsymbol{w}$ crosses from $\mathcal{S}$ to $\overline{\mathcal{S}}$ or vice versa (*zero-cross*). This ensures that if the walks have succeeded in finding a sparse cut, the matrix update does not add any mass across it.

Algorithm 18: cutConstruction

Result: $\mathcal{S} \subseteq \mathcal{V}^*$, $\mathcal{WS}$
Input: $\mathcal{G}^*$: input graph ;
b: number of walk iterations per batch ;
k: minimum number of nodes needed to be seen by initialization walks ;
ℓ: maximum number of walks per iteration ;

Sample initial disjoint walks:
Sample seed sets $\mathcal{S}_1$, $\mathcal{S}_2$ and walks $\boldsymbol{q}[1]$, $\boldsymbol{q}[2]$ using Algorithm 17 ;
Initialize votes:
foreach $v \in \mathcal{V}^*$ **do**
  if $v \in \mathcal{S}_1$ **then**
    $g(v) = 1$;
  else if $v \in \mathcal{S}_2$ **then**
    $g(v) = -1$;
  else
    $g(v) = 0$;
  end
end
Build list of walk lists:
$\mathcal{WS} = \emptyset$, marked nodes $V' = \mathcal{S}_1 \cup \mathcal{S}_2$;
if $V' = \mathcal{V}^*$ **then**
  Let $\mathcal{WS} = [[\boldsymbol{q}[1], \boldsymbol{q}[2]]]$. Return $\mathcal{WS}$ and $\mathcal{S} = \mathcal{S}_1$;
for $j \leftarrow 1$ **to** b **do**
  Initialize list $\mathcal{W}_j = [\boldsymbol{q}[1], \boldsymbol{q}[2]]$;
  while $V' \neq V$ *and* $|\mathcal{W}_j| < \ell$ **do**
    Choose random node v from V' with 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 $\boldsymbol{w}(t) \notin \mathcal{S}_1 \cup \mathcal{S}_2$ **do**
      $t = t + 1$;
      $\boldsymbol{w}(t)$ sampled using the standard random walk (Section 2.4) ;
    end
    for $i \leftarrow 1$ **to** $t - 1$ **do**
      Add $\boldsymbol{w}(i)$ to V' ;
      if $\boldsymbol{w}(t) \in \mathcal{S}_1$ **then**
        $g(\boldsymbol{w}(i)) = g(\boldsymbol{w}(i)) + 1$;
      else
        $g(\boldsymbol{w}(i)) = g(\boldsymbol{w}(i)) - 1$;
      end
      Append $\boldsymbol{w}$ to $\mathcal{W}_j$;
    end
  end
  Append $\mathcal{W}_j$ to $\mathcal{WS}$;
end
Construct cut using votes:
$\mathcal{S} = \{v \in \mathcal{V}^* \ : \ g(v) > 0\}$;
Return $\mathcal{S}$ and $\mathcal{WS}$;

Algorithm 19: MatrixUpdate

Result: Frequency matrix $\tilde{A}$ updated for pairs of nodes in walks in $\mathcal{WS}$
Input: $\tilde{A}$: frequency matrix ;
$\mathcal{S}$: cut ;
$\mathcal{WS}$: list of walk lists ;
zeroCross: flag that says whether or not to add mass across cuts ;
walkDistWeight: flag that says whether or not to weight the amount of mass added by the number of steps between nodes ;
maxDist: maximum steps between nodes on walks for which to update ;

---

for $\mathcal{W} \in \mathcal{WS}$ **do**
  for $\boldsymbol{w} \in \mathcal{W}$ **do**
    if $\boldsymbol{w}(1) \in \mathcal{S}$ **then**
      Let ℓ' be the smallest integer such that $\boldsymbol{w}(\ell'+1) \notin \mathcal{S}$ if such a ℓ' exists and $length(\boldsymbol{w})$ otherwise ;
    else
      Let ℓ' 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* **then**
      $\ell = \min(\ell', \text{maxDist})$;
    else
      $\ell = \min(length(\boldsymbol{w}), \text{maxDist})$;
    for $z \leftarrow 1$ **to** $\ell - 1$ **do**
      if *walkDistWeight* **then**
        $q = 1/z$;
      else
        $q = 1$;
      for $t = 1, 2, \ldots \ell - z$ **do**
        $u = \boldsymbol{w}(t)$ 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
  end
  Set $\tilde{a}_{v,v} = 0$ for all $v \in \mathcal{V}^*$;
end

The variants we try for matrix updating are summarized in the list below:

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

9.6 Random walk-based generation in context of non-random measures

Random walk-based generation assigns $\tilde{a}_{v,u}$ with respect to the following notions of "closeness" between nodes v and u:

1. The number of times the nodes appear on the same walk.
2. The inverse of the number of steps between the nodes.
3. Whether or not the nodes appear on the same side of the cut discovered in the cut construction phase.

In order to contextualize how well these closeness notions produce useful frequency matrices, we compare Random walk-based generation frequency matrices to matrices computed from non-random measures that are related to these closeness notions.

The walk distance weights in the matrix update phase are akin to a weight computed from shortest path distance. The shortest path distance $\text{sp}(v, u, \mathcal{G}^*)$ between nodes v and u in $\mathcal{G}^*$ 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 the average number of steps between them on these walks is likely proportional to their shortest path distance. The walk distance weight rule assigns weight inversely proportional to the number of steps between pairs on walks. Thus, 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 $\widetilde{SPM}(k)$ with inverse powers k so that $\widetilde{spm}(k)_{u,v} = (1/sp(u, v, \mathcal{G}^*))^k$.

By using random walks, it seems natural to assume that pairs (v, u) that have small *average commute time* would appear more often and closer together on the same walks than those that do not. Let $c'(v \mid u, \mathcal{G}^*)$ be the expected number of steps a standard random walk starting at v takes before hitting u for the first time and returning again to v. This quantity is not symmetric. The average commute time of a pair (v, u) in $\mathcal{G}^*$ is $ct(v, u, \mathcal{G}^*) = c'(u \mid v, \mathcal{G}^*) + c'(v \mid u, \mathcal{G}^*)$ so that $ct(v, u, \mathcal{G}^*) = ct(u, v, \mathcal{G}^*)$. As we expect the frequency that any pair of vertices v and u appear on the same walk to be decreasing in commute time, we compute $\widetilde{CTM}$ with inverse power k so that $\widetilde{ctm}_{v,u} = (1/ct(v, u, \mathcal{G}^*))^k$. 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 compare probabilistic adjacency matrices 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 adjacency matrix so that we can compare the utility of these probabilistic adjacency matrices to those generated using Random walk-based generation.

9.7 Experiments

We focus our experiments on

1. How does batch size affect the sparsity of the cuts discovered in Algorithm 18?

2. How do different matrix update rules in Algorithm 19 affect the expected spectrum of $\mathcal{G}$ drawn using independent edge sampling on A in expectation? As discussed in Section 5.1, the spectrum of $\mathcal{G}$ is concentrated around $\boldsymbol{\lambda}(L(A))$ (Theorem 5.1.2). Therefore, we use $\boldsymbol{\lambda}(L(A))$ to track how well Random walk-based generation is learning to model $\mathcal{G}^*$ by computing $\ell_2^{\mathrm{LW}}(\boldsymbol{\lambda}(L(A)), \boldsymbol{\lambda}^*)$.

3. Can the structure revealed by Random walk-based generation be recovered using shortest paths and commute times alone? How do the probabilistic adjacency matrices computed using the shortest path model and commute time model compare to A generated by Random

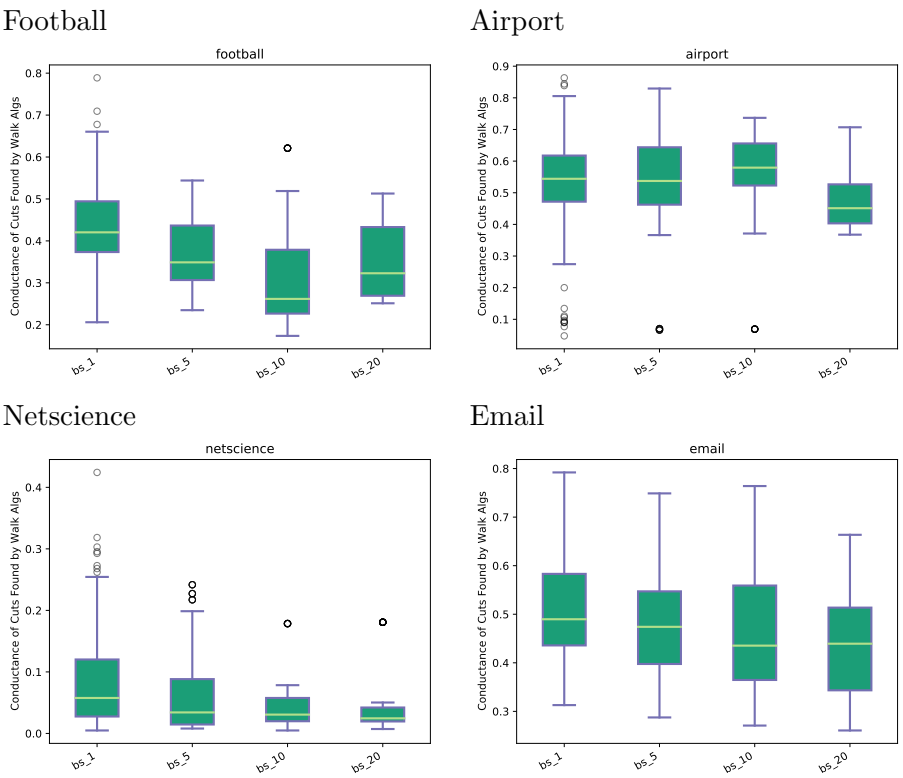


Figure 9.1: Conductance of cuts found via Random walk-based generation. We observe that a larger batch size finds cuts with smaller conductance on average.

walk-based generation?

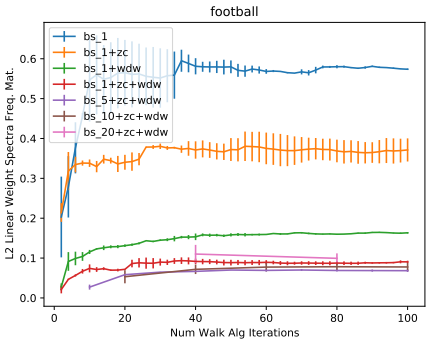
We experiment with different batch sizes to see how large the batch size should be to help produce sparse cuts (Table 9.1). We observe that increasing the batch size seems to slightly help to discover cuts with smaller conductance.

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).

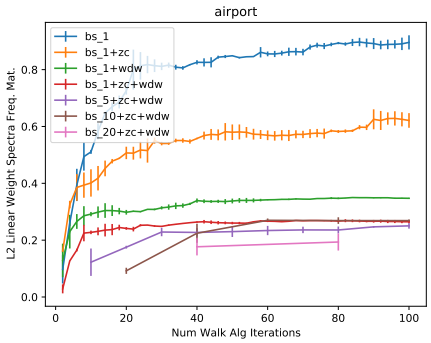
We plot the eigenvalues of A across our various hyper-parameters and shortest path models and commute time models (SPM and CTM) for a fixed k (Figure 9.4). As k grows, SPM and CTM capture more spectral structure (Figure 9.5). This is likely because as k grows, the weights for pairs (v, u) with longer 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 $\ell_2^{\text{LW}}(\boldsymbol{\lambda}^*, \boldsymbol{\lambda}(L(SPM)))$

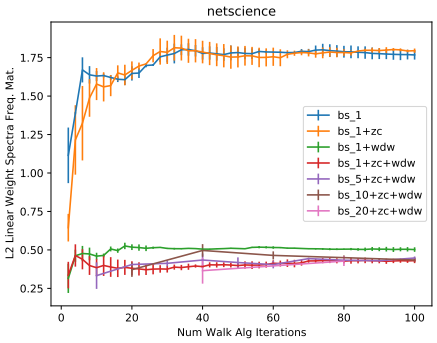
Football



Airport



Netscience



Email

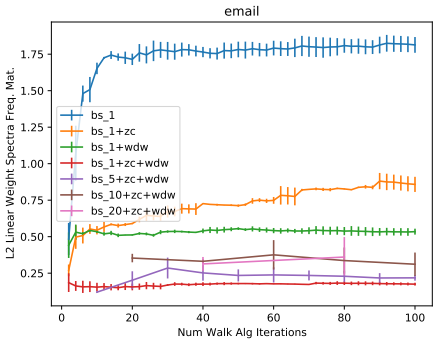
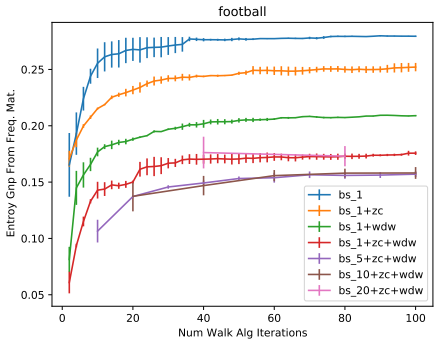
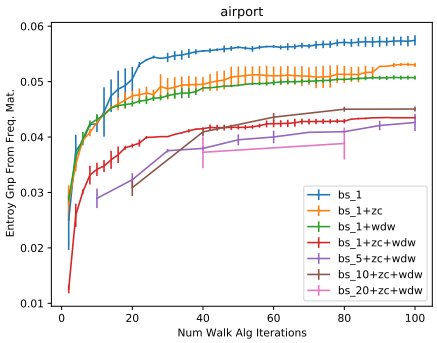


Figure 9.2: Difference between true spectrum and spectrum of probababilistic adjacency matrix A measured by ℓ_2^{LW} after $T = 100$ sets of walks for different Random walk-based generation parameters.

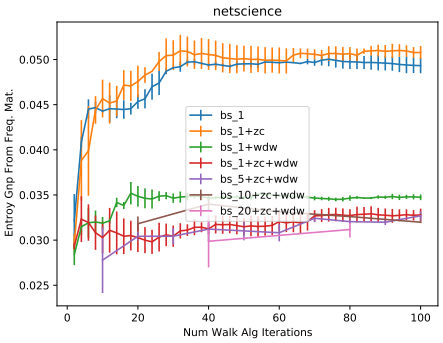
Football



Airport



Netscience



Email

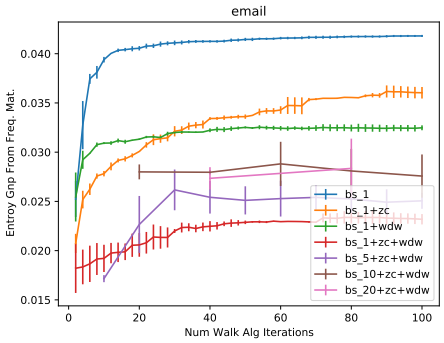
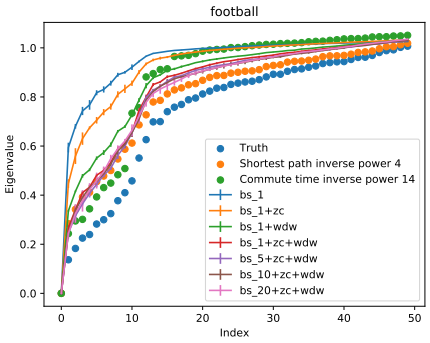
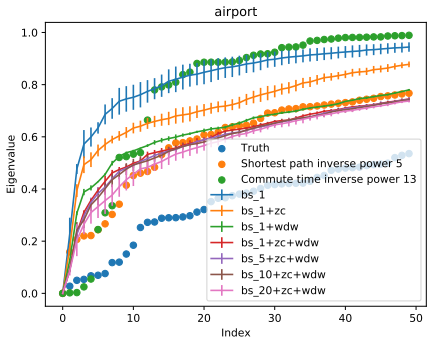


Figure 9.3: Mean entropy of probababilistic adjacency matrix entries $a_{u,v}$ after $T = 100$ sets of walks for different Random walk-based generation parameters.

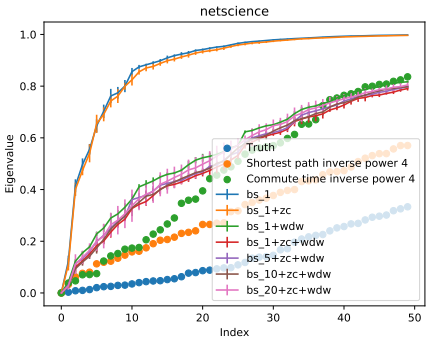
Football



Airport



Netscience



Email

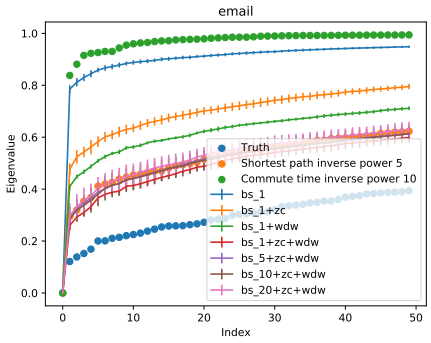


Figure 9.4: Eigenvalues of probababilistic adjacency matrices after 100 walk algorithm iterations compared to inverse of shortest path matrix.

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

Table 9.1: 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

Table 9.2: Entropy of commute time model CTM with entries $ctm_{u,v} = (1/CT(\mathcal{G}^*, u, v))^k$.

is at least as small as $\ell_2^{\text{LW}}(\boldsymbol{\lambda}^*, \boldsymbol{\lambda}(L(A)))$ 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 (Tables 9.1, 9.2). 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 9.5). For the Netscience graph, the smallest k for CTM and SPM that matched 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. Furthermore, the task of finding k does not seem straightforward, especially for CTM for which it varied drastically for the input graphs we tried. For example, to match the performance of Random walk-based generation with 100 walk algorithm iterations, $k = 4$ was needed for the Netscience graph but $k = 13$ was needed for the Airport graph.

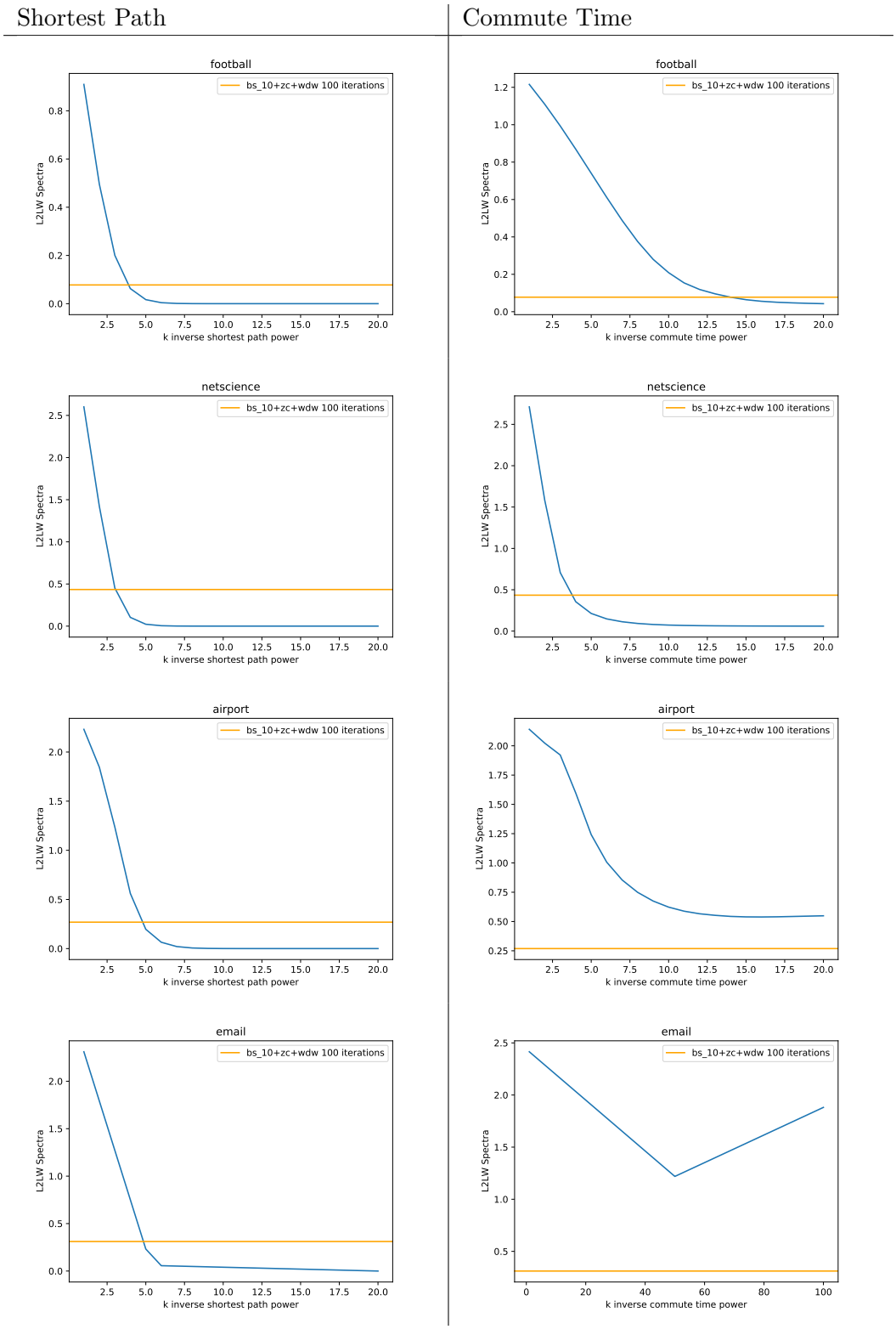


Figure 9.5: Difference between true spectrum and spectrum of shortest path and commute time probabilistic adjacency matrices measured by ℓ_2^{LW} for growing inverse power k. The average spectral performance for Random walk-based generation with walk-distance weight and zero cross updates with a batch size of 10 is plotted for reference.

Chapter 10

Cut fix generation: graph generation by matching cut connectivity

Connectivity across cuts is one of the most central global graph features explored throughout this work that capture graph structure (cut connectivity is defined in Definition 5.1.1). In fact, matching the connectivity in A^* across all cuts characterizes $\mathcal{G}^*$ exactly (shown in Claim 5.1.1). 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. Spectrum-matching generation (Chapter 7), the NetGAN with the fastest mixing Markov chain (Chapter 8), and Random walk-based 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 (Erdős and Rényi, 1960). By contrast, Cut fix generation chooses cuts by starting with a seed probabilistic adjacency matrix A_0. It then repeatedly identifies cuts with approximately the largest difference in total edge weight across the cut, compared to A^*. Once a cut $\mathcal{S}$ has been identified, it is "corrected" by making fractional edge additions/deletions so that the connectivity across $\mathcal{S}$ matches the connectivity in A^*. The idea is that once enough cuts have been corrected, the final $\mathcal{G}$ generated from independent edge sampling (Definition 2.3.1) should have structure similar to $\mathcal{G}^*$.

Recall from Definition 5.1.1 that the connectivity of $\mathcal{S}$ with respect to A measures the sum of probabilistic edge weights 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 $\ell_2^{\text{LW}}(\lambda(A^*), \lambda(A))$ or some other metric. Once A is corrected, $\mathcal{G}$ is generated by

including each pair (v, u) in $\mathcal{E}$ with probability $a_{v,u}$.

To see why adaptively choosing cuts might be helpful, consider a modified barbell graph as $\mathcal{G}^*$ that has two dense clusters $\mathcal{S}^*$ and $\overline{\mathcal{S}}^*$ connected by a single edge. For any A, if the connectivity of $\mathcal{S}^*$ is corrected to have a single edge crossing and the remaining edge mass is placed on $\mathcal{S}^* \times \mathcal{S}^*$ and $\overline{\mathcal{S}}^* \times \overline{\mathcal{S}}^*$ then A becomes a reasonable model of $\mathcal{G}^*$.

A cut correction algorithm requires the following:

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

2. A method for constructing 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 $\mathcal{G}$ is drawn from.

3. The number of cuts to correct. If too many are corrected, there will be little variability among $\mathcal{G}$.

The Cut fix generation algorithm is described in Algorithm 20.

Algorithm 20: Cut Fix Generation.

Result: Probabilistic adjacency matrix A with several cuts altered to have connectivity that matches A^*.
Input: A^*: input adjacency matrix ;
A: probabilistic adjacency matrix ;
k: number of cuts to correct ;
α: neighborhood size for $GRASP$ cut sampling ;

---

for $i \leftarrow 1$ **to** k **do**
  $\mathcal{S} = approxGrasp(A^*, A, \alpha)$ (Algorithm 26) ;
  $A = cutCorrect(A^*, A, \mathcal{S})$ (Algorithm 27) ;
end

Cut fix generation corrects cuts with connectivity that differs the most between A and A^*. This step is further explained in Section 10.2. These cuts are an approximate solution to max $|\partial(\mathcal{S}, A) - \partial(\mathcal{S}, A^*)|$ using a *Greedy Randomized Adaptive Search Procedure* (GRASP, the general framework can be found in (Feo and Resende, 1995)) 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 an $\mathcal{S}$ that approximately 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 positive and negative edge weights.

Once an $\mathcal{S}$ is chosen to correct, it is corrected in A using a symmetric perturbation matrix E. This step is further explained in Section 10.3. The matrix E is constructed so that $\partial(\mathcal{S}, A + E) = \partial(\mathcal{S}, A^*)$ and $a_{v,u} + e_{v,u} \in [0, 1]$ for all v, u. 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 $\mathcal{G}$ has approximately m^* edges in expectation. We construct E with $\sum_{u,v} e_{v,u} = 0$ so that if the seed matrix A has $\sum_{u,v} a_{v,u} = m^*$, then the expected number of edges m^* in $\mathcal{G}$ 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) provides a relationship between the largest cut connectivity difference between two graphs (cut distance) and the number of adjacency preserving vertex maps between the two graphs, another notion of graph similarity.

Cut fix generation is related to a broader generative graph model technique called *editing* which takes a baseline graph and then makes corrections, usually in the form of edge additions and deletions, in a way that generates realistic graphs. Perhaps the most classical example of an editing generative graph model is edge swapping 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. A more detailed description of MUSKEETER is in Section 9.3.

A large part of the Cut fix generation algorithm is identifying cuts to correct. We identify cuts that have a large absolute value in a weighted graph with positive and negative edge weights (Section 10.2). This is related to the approximation algorithms designed to find the maximum weighted cut in a graph. One algorithm to find the cut with maximum weight is an approximation scheme by Goemans and Williamson (1995) that uses semidefinite programming. We experimented with variations of this approximation algorithm for identifying cuts to correct. Ultimately, the algorithms described in Section 10.2 were faster than the semidefinite programming algorithms we tried.

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 maximizer 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), typically random ;

    solution = localSearch(solution, inputInstance), typically random ;

    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 a locally optimal solution is found — a solution for which all of its neighbors have smaller or equal objective value to 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 maximum 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 entrywise difference between A and A^*.

The greedy construction phase begins with an initialization stage. Nodes $\mathcal{V}^*$ are permuted uniformly at random. The first $\beta \cdot n^*$ nodes in the order of that permutation are randomly placed according to independent random bits into two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ each with probability $\frac{1}{2}$ where

$\beta \in [0,1]$. The sets $\mathcal{S}_1$ and $\mathcal{S}_2$ initialize[1] $\mathcal{S}$ and $\overline{\mathcal{S}}$. The remaining nodes $v \in \mathcal{V}^* \setminus (\mathcal{S}_1 \cup \mathcal{S}_2)$ are placed 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 and $\mathcal{G}$)**.** Let W be an $n^* \times n^*$ real symmetric matrix, $\mathcal{S} \subseteq \mathcal{V}^*$, and $v \in \mathcal{V}^*$. Let $\mathcal{G}$ be a directed or undirected graph on n^* vertices. 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 done in the initialization step, the remaining nodes are placed greedily by uniformly at random permuting $\mathcal{V}^* \setminus (\mathcal{S}_1 \cup \mathcal{S}_2)$. Nodes $v \in \mathcal{V}^*$ are considered in the permutation order: if $|f(v, \mathcal{S}_1, W)| \geq |f(v, \mathcal{S}_2, W)|$ then $\mathcal{S}_2 = \mathcal{S}_2 \cup \{v\}$. Otherwise, $\mathcal{S}_1 = \mathcal{S}_1 \cup \{v\}$. Node v is placed in the set to which it is less connected in order to greedily maximize the absolute value of the connectivity across the cut. Once all nodes are placed, $\mathcal{S}_1$ is renamed to $\mathcal{S}$. The details of *GRASP* greedy-construction are in Algorithm 24 using the approximation variant we explain in Section 10.2.3.

Using the 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 iteration of local search steps, $\mathcal{V}^*$ is permuted uniformly at random and visited in permutation order. If $v \in \mathcal{S}$, $\mathcal{S}' = \mathcal{S} \setminus \{v\}$ is considered. Otherwise, $\mathcal{S}' = \mathcal{S} \cup \{v\}$ is considered. If $|\partial(\mathcal{S}', W)| > |\partial(\mathcal{S}, W)|$, then $\mathcal{S}$ is updated to $\mathcal{S}'$. After all the nodes are visited, if no changes were made to $\mathcal{S}$ or a maximum number of search steps is reached then the search is terminated. If any changes were made to $\mathcal{S}$, another iteration of local search steps is performed. The details of *GRASP* local search are in Algorithm 25 using the approximation variant we explain in Section 10.2.3.

Greedy construction and local search are repeated K times, keeping track of the set $\mathcal{S}$ with the largest $|\partial(\mathcal{S}, W)|$. After K iterations, the $\mathcal{S}$ with the largest $|\partial(\mathcal{S}, W)|$ is output (Algorithm 26).

[1]Nodes are added to $\mathcal{S}_1$ and $\mathcal{S}_2$ until they form a partition of $\mathcal{V}^*$, at which point $\mathcal{S} = \mathcal{S}_1$.

10.2.3 Approximate $GRASP$: Sub-neighborhood

In order to scale to larger graphs, we introduce a $GRASP$ variant which looks at only a subset 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. The computational bottleneck is the repeated computation of cut connectivities. To reduce this computational burden, we introduce an approximation that uses only a subset of nodes for each $v \in \mathcal{V}^*$ to compute the connectivity of v to any $\mathcal{S}$. Most of the A considered are non-zero almost everywhere, resulting in W non-zero everywhere, so most nodes in $\mathcal{V}^*$ affect the connectivity of v. However, most of the entries in A are near zero, so the subset is sampled to prioritize nodes $u \in \mathcal{V}^*$ with large $|w_{v,u}|$ that have higher impact on the connectivity of v.

Rewriting connectivity in terms of vertex connectivity, $\partial(\mathcal{S}, W) = \sum_{v \in \overline{\mathcal{S}}} f(v, \mathcal{S}, W)$. For each node v, instead of keeping track of all n entries of $\mathbf{w}_v$ where $\mathbf{w}_v$ is the v-th row of W, the algorithm keeps track of a random subset of size αn^* where $\alpha \in (0, 1)$. This subset is called the *outgoing neighbors* of v, written as $\mathcal{N}_{OUT}(v)$. The vertex connectivity of v to $\mathcal{S}$ is computed using only $\mathcal{N}_{OUT}(v)$ so $f(v, \mathcal{N}_{OUT}(v) \cap \mathcal{S}, W)$. The step where a subset of neighbors is sampled for each node is called *neighborhood subset sampling* (Algorithm 23). Neighborhood subset sampling outputs a directed graph $\mathcal{G}'$ in which the out-degree of each node is αn^*. The directed graph $\mathcal{G}'$ encodes the incoming and outgoing neighbors for each vertex.

Algorithm 23: Neighborhood subset sampling.

Result: Directed graph $\mathcal{G}'$
Input: W: weight matrix of dimension $n^* \times n^*$;
$\alpha \in (0, 1)$: neighborhood sample fraction ;

Initialize directed graph $\mathcal{G}'$ with nodes $\mathcal{V}^*$ and without edges ;
for $v \leftarrow 1$ **to** n^* **do**
  Sample a 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 proportional to $|w_{v,u}|$;
  For all $u \in \mathcal{N}_{OUT}(v)$, insert a directed edge from v to u in $\mathcal{G}'$;
end

The reason this is useful is to prevent updating the connectivity of all nodes each time a vertex is removed from/added to $\mathcal{S}$ as part of the local search. Because W is usually dense, the connectivity of nearly every node to $\mathcal{S}$ will change with each removal from/addition to $\mathcal{S}$. With neighborhood

Algorithm 24: Approximate *GRASP* greedy construction.

Result: Cut $\mathcal{S}$;
Connectivity lists for $\mathcal{S}$ and $\overline{\mathcal{S}}$ denoted by $\mathbf{s}$ and $\overline{\mathbf{s}}$;
Input: W: weight matrix ;
$\mathcal{G}'$: directed graph with neighbor lists $\mathcal{N}_{OUT}(v)$ for each v;
β: *GRASP* initialization size ;

Initialize sets $\mathcal{S}_1 = \mathcal{S}_2 = \emptyset$;
Randomly place βn^* 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 π sampled uniformly at random that maps $[n^*]$ to $\mathcal{V}^*$;
for $i = 1, 2, \ldots, \beta n^*$ **do**
  Place $\pi(i)$ into either $\mathcal{S}_1$ or $\mathcal{S}_2$ with independent fair coin flip ;
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 π sampled uniformly at random that maps $[(1-\beta) \cdot n^*]$ to $\mathcal{V}^* \setminus (\mathcal{S}_1 \cup \mathcal{S}_2)$;
for $i \leftarrow 1$ **to** $(1-\beta) \cdot n^*$ **do**
  $v = \pi(i)$;
  if $|s_v| \geq |\overline{s}_v|$ **then**
    $\mathcal{S}_2 = \mathcal{S}_2 \cup \{v\}$;
  else
    $\mathcal{S}_1 = \mathcal{S}_1 \cup \{v\}$;
  for $u \in \mathcal{N}_{IN}(v)$ **do**
    if $v \in \mathcal{S}_1$ **then**
      $s_u = s_u + w_{v,u}$;
    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 ;
Input: W: weight matrix ;
$\mathcal{G}'$: directed graph ;
$\mathcal{S}$ and $\overline{\mathcal{S}}$: cut and cut complement ;
$\mathbf{s}$ and $\overline{\mathbf{s}}$: connectivity lists ;
K: number of grasp iterations ;
L: length of candidate list history to detect cycles ;
Set $x = \partial(\mathcal{S}, W)$ and initialize seen candidate lists to an empty list ;
Candidate lists is a list of lists where each list is a list of candidates. Used to detect cycles ;

---

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 uniformly at random permutation π ;
  Consider local change for each candidate in permutation π 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* **then**
      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 large absolute connectivity in W;
Input: W: weight matrix ;
$\beta \in (0,1)$: $GRASP$ initialization size ;
$\alpha \in (0,1)$: $GRASP$ neighborhood fraction size ;
K: number of grasp iterations ;
L: candidate list history to detect cycles ;

$\mathcal{G}'$ = neighborhoodSubsample(W, α) ;
for $\ell \leftarrow 1$ **to** L **do**
  $\mathcal{S}$', $\mathbf{s}$, $\bar{\mathbf{s}}$ = greedyConstruction(W, $\mathcal{G}'$, β) (Algorithm 24) ;
  $\mathcal{S}$' = localUpdate(W, $\mathcal{G}'$, $\mathcal{S}$', $\mathbf{s}$, $\bar{\mathbf{s}}$, K, L) (Algorithm 25);
  if $\ell = 0$ *or* $|\partial(\mathcal{S}', W)| > best$ **then**
    best = $|\partial(\mathcal{S}', W)|$;
    $\mathcal{S} = \mathcal{S}$' ;
end

subset sampling, only the connectivities 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 vertices are added/removed, the set of nodes to update when adding/removing v is its *incoming neighbors* $\mathcal{N}_{IN}(v)$ which is the set of nodes u such that $v \in \mathcal{N}_{OUT}(u)$.

However, many of the entries are near-zero so that they have little effect on the connectivity of any cut. With this in mind, $\mathcal{N}_{OUT}(v)$ is sampled 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 a perturbation

Once a cut $\mathcal{S}$ to correct is found, a symmetric perturbation matrix E is constructed such that $\partial(\mathcal{S}, A+E) = \partial(\mathcal{S}, A^*)$ and the entries of $B = A + E$ satisfy $b_{v,u} \in [0,1]$. Matrix E is constructed in two stages (1) node pair partition assignment (2) node partition update.

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

Definition 10.3.1 (Node pair partition $\mathcal{Q}_{\mathcal{S}}$ induced by $\mathcal{S}$)**.** For $\mathcal{S} \subseteq \mathcal{V}^*$, the node pair partition induced by $\mathcal{S}$ is $\mathcal{Q}_{\mathcal{S}} = \langle Q_{\mathcal{S}}, Q_{\overline{\mathcal{S}}}, Q_{\mathcal{S},\overline{\mathcal{S}}} \rangle$ where

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

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

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

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 the matrix X of dimension $\mathcal{V}^* \times \mathcal{V}^*$ is a triple $T(X, \mathcal{Q}_{\mathcal{S}}) = \langle t(X, Q_{\mathcal{S}}), t(X, Q_{\overline{\mathcal{S}}}), t(X, Q_{\mathcal{S},\overline{\mathcal{S}}})\rangle$. 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.

Our goal is to find a feasible *assignment* on $\mathcal{Q}_{\mathcal{S}}$ to construct 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 $T(A^*, \mathcal{Q}_{\mathcal{S}}) = \langle t(A^*, Q_{\mathcal{S}}), t(A^*, Q_{\overline{\mathcal{S}}}), t(A^*, Q_{\mathcal{S},\overline{\mathcal{S}}})\rangle$ which is the number of edges on $\mathcal{Q}_{\mathcal{S}}$ in $\mathcal{G}^*$.

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 $\mathcal{S}$ in $A + E$. 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 $\mathcal{S}$ under A^*. Feasible $Y(\mathcal{Q}_{\mathcal{S}})$ under the *triple* constraint imposes two additional requirements. For the triple constraint, $y(Q_{\mathcal{S},\overline{\mathcal{S}}}) = t(A^*, Q_{\mathcal{S},\overline{\mathcal{S}}})$, $y(Q_{\mathcal{S}}) = t(A^*, Q_{\mathcal{S}})$, and $y(Q_{\overline{\mathcal{S}}}) = t(A^*, Q_{\overline{\mathcal{S}}})$ so there is only one feasible assignment $Y(\mathcal{Q}_{\mathcal{S}})$. For the single constraint, the optimal assignment over the set of all feasible $Y(\mathcal{Q}_{\mathcal{S}})$ is chosen by minimizing the changes to the assignment $T(A, \mathcal{Q}_{\mathcal{S}})$. In addition to matching the connectivity across $\mathcal{S}$, assignment $y(Q)$ is constrained to be non-negative and not exceed the number of pairs in Q for each $Q \in \mathcal{Q}_{\mathcal{S}}$ so that E can be constructed with entries of $A + E$ within $[0, 1]$. To match the number of edges in $\mathcal{G}^*$ in expectation, $Y(\mathcal{Q}_{\mathcal{S}})$ is constrained to sum up to m^*. The set of feasible assignments $Y(\mathcal{Q}_{\mathcal{S}})$ can be written as a small linear program (Equation 10.2) with only three variables. The objective is quadratic to minimize perturbations to $T(A, \mathcal{Q}_{\mathcal{S}})$.

$$\text{Minimize} \quad \sum_{Q \in \mathcal{Q}_\mathcal{S}} (y(Q) - t(A, Q))^2 \tag{10.2}$$

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

Once the algorithm has found satisfying assignment $Y(\mathcal{Q}_\mathcal{S})$, each set of pairs $Q \in \mathcal{Q}$ can be treated separately. For each $Q \in \mathcal{Q}_\mathcal{S}$, the assignment $y(Q)$ is used 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 $E^{(Q)}$ which we call *uniform* and *push*. In both methods, $E^{(Q)} = 0_{n^*}$ initially. If $y(Q) - t(A, Q)$ is negative, positive value is removed from entries of $E^{(Q)}$. If $y(Q) - t(A, Q)$ is positive, then positive value is added to entries of $E^{(Q)}$. For both uniform and push, value is removed from (added to) $e^{(Q)}_{v,u}$ pairs $(v, u) \in Q'$ where $Q' \subseteq Q$ is constructed by the uniform or push algorithm. The amount $\delta_{Q'}$ removed from (added to) $e^{(Q)}_{v,u}$ for pairs $(v, u) \in Q'$ is the minimum $a_{v,u}$ (minimum $(1 - a_{v,u})$) so that $a_{v,u} + e^{(Q)}_{v,u}$ is at least 0 (at most 1) for all $(v, u) \in Q'$. To guarantee progress, Q' is all pairs (v, u) with $a_{v,u}$ at least ϵ (at most $1 - \epsilon$) so that $\delta_{Q'} \geq \epsilon$ when removing (adding). For the uniform method, this is the only constraint on Q'.

The push methods aims to remove (add) from low (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 ϵ (below $1 - \epsilon$) so $\delta_{Q'} \geq \epsilon$ when removing below (adding above) k to remove relatively small (add relatively large) values. The median k defines which values are small (large), but needs to be updated if the current k restricts $Q' = \emptyset$ so there are not any pairs for which value can be removed (added).

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 $E^{(Q)}$ is equal to $y(Q) - t(A, Q)$. If Q' is empty, the constraints on Q' are relaxed to

include more pairs. If using the uniform update, ϵ is divided by 2 to relax the constraint. For push, a constant b is added (removed) to k until k reaches 1 or 0. At that point, ϵ is divided by 2. The entire cut correction algorithm is in Algorithm 27.

Algorithm 27: Cut correction

Result: Probabilistic adjacency matrix A that matches the connectivity across $\mathcal{S}$ to A^*
Input: A: probabilistic adjacency matrix of dimension $n^* \times n^*$;
A^*: target graph adjacency matrix ;
$\mathcal{S} \subseteq \mathcal{V}^*$: cut ;
typeConstraint which takes value single or triple ;
typeUpdate which takes value uniform or push ;
ϵ: progress parameter ;
b: constant to relax feasible pair constraints ;

Compute node pair 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 triple of node 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 Experiments

We run experiments for both seed probabilistic adjacency matrices generated using Random walk-based generation and the *uniform* seed A_0 which has entries $a_{v,u} = m^*/(n^2 - n)$ for all $v \neq u$. As we further discuss in Section 10.4.3, Random walk-based generation and Cut fix generation run faster than NetGAN and Spectrum-matching generation so we ran experiments for not only the graphs used for those methods but larger graphs as well. All graphs used are described in Chapter 6.

10.4.1 Trade off between entropy and matching spectra

We ran extensive experiments for both triple and single constraints with push updates. Ideally, correcting cuts in A should help match $\boldsymbol{\lambda}(L(A))$ to $\boldsymbol{\lambda}^*$. We found that across all graphs, correcting

Algorithm 28: Construct Perturbation Matrix.

Result: Perturbation matrix E
Input: A: probabilistic adjacency matrix of dimension $n^* \times n^*$;
A^*: target graph adjacency matrix ;
$\mathcal{S} \subseteq \mathcal{V}^*$: cut ;
typeConstraint which takes value single or triple ;
typeUpdate which takes value uniform or push ;
ϵ: progress parameter ;
b: constant to relax feasible pair constraints ;

$E = 0_{n^*}$;
for $Q \in \mathcal{Q}_{\mathcal{S}}$ **do**
  $E(Q)$ is the sub-matrix E of entries $e_{v,u}^{(Q)}$ of $(v,u) \in Q$;
  k is the median of all entries in $E(Q)$ above ϵ ;
  while $\sum_{(v,u)\in Q} e_{v,u}^{(Q)} < y(Q) - t(A,Q)$ **do**
    if *typeUpdate is uniform* **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**
        $Q' = \{(v,u) \in Q \ : \ e_{v,u} < 1 - \epsilon \text{ and } e_{v,u} > k\}$
      else
        $Q' = \{(v,u) \in Q \ : \ e_{v,u} > \epsilon \text{ and } e_{v,u} < k\}$
    if $Q' = \emptyset$ **then**
      if *typeUpdate is uniform* **then**
        $\epsilon = \frac{1}{2}\epsilon$
      else
        if $y(Q) - t(A,Q) > 0$ **then**
          $k = \max(0, k - b)$;
          If $k = 0$, then $\epsilon = \frac{1}{2}\epsilon$;
        else
          $k = \min(1, k + b)$;
          If $k = 1$, 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**
        $a_{v,u} = a_{v,u} + \text{sign} \cdot \delta(Q')$, $a_{u,v} = a_{v,u}$;
      end
  end
end

cuts helps match spectra measured by ℓ_2^{LW} (Figures 10.1, 10.2, and 10.3). This comes at an entropy cost (Figures 10.4, 10.5, and 10.6). The entropy decrease happens rapidly for cut corrections with a Random walk-based generation seed matrix.

Unsurprisingly, the Random walk-based generation seeds outperform the uniform seed for a small number of corrected cuts (around 15 or fewer). However, the uniform seed does perform comparably for most inputs once enough cuts have been corrected (around 30) and is much quicker to construct (Section 10.4.3). For some inputs, the uniform seed is significantly inferior to the Random walk-based generation seeds in matching the spectrum, regardless of how many cuts are corrected (e.g., ROME and HEALTH).

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 10.7). While the uniform updates do improve spectral performance on the uniform seed probabilistic adjacency matrices, the improvements are much slower than with the push updates. The single constraint performance is comparable for Random walk-based generation seed matrices, but the triple constraint more reliably improves the spectral performance with each cut corrected. The opposite is true for the uniform seed, with the triple constraint nearly matching the performance of the single constraint but reliably the single constraint does better. The reason why is unclear, it seems as though the triple constraint should always be advantageous for matching the properties of $\mathcal{G}^*$ because in addition to matching the number of edges crossing the cut it matches the number on either side (regardless of the seed probabilistic adjacency matrix).

10.4.3 Time of Cut fix generation compared to Random walk-based generation

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

It takes under 17 hours to sample a Random walk-based generation probabilistic adjacency matrix as a seed matrix with 50 random walk iterations and correct 25 cuts for our largest input

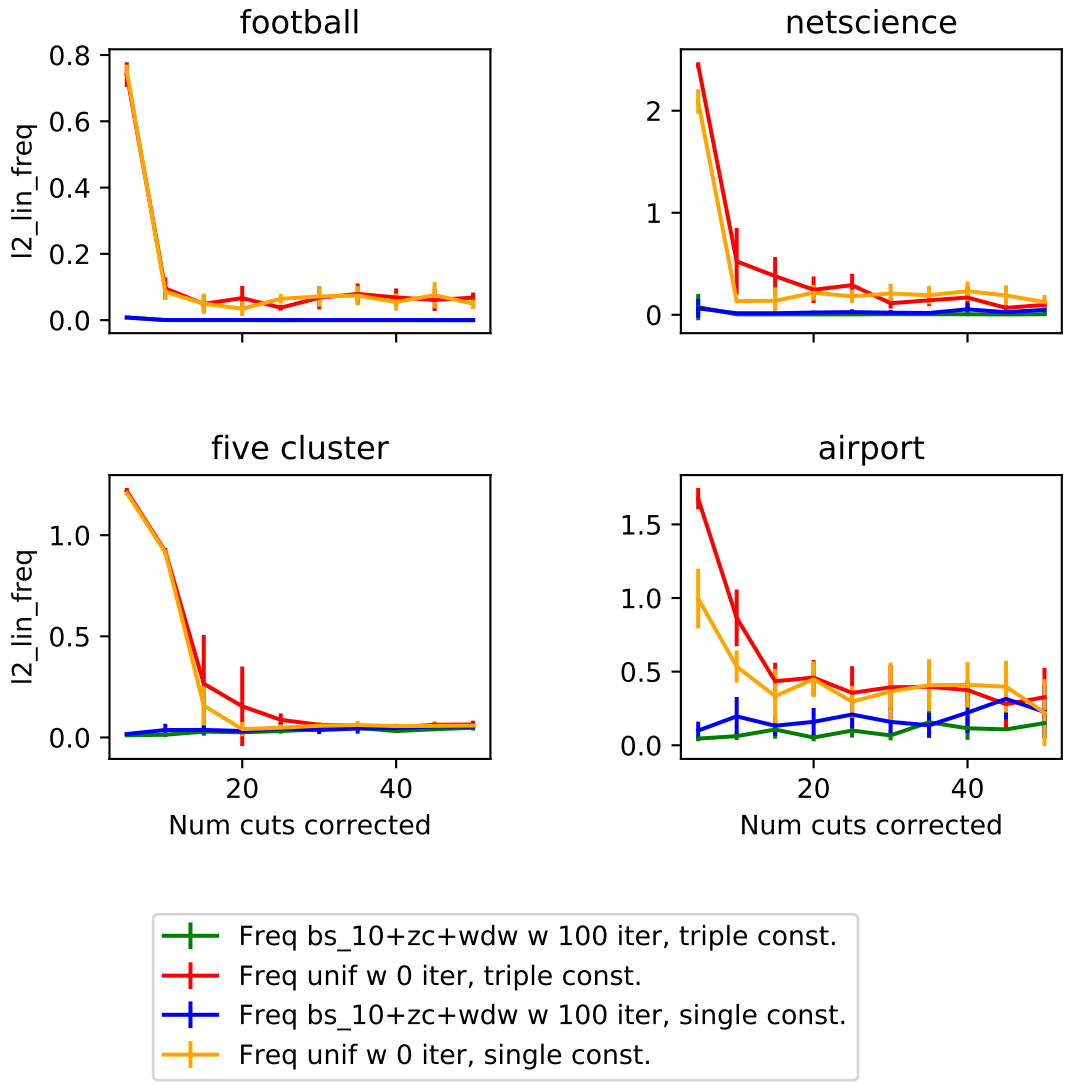


Figure 10.1: Number of cuts corrected against ℓ_2^{LW} ($\boldsymbol{\lambda}^*$,$\boldsymbol{\lambda}$) for Football, Netscience, Five Cluster, and Airport graphs.

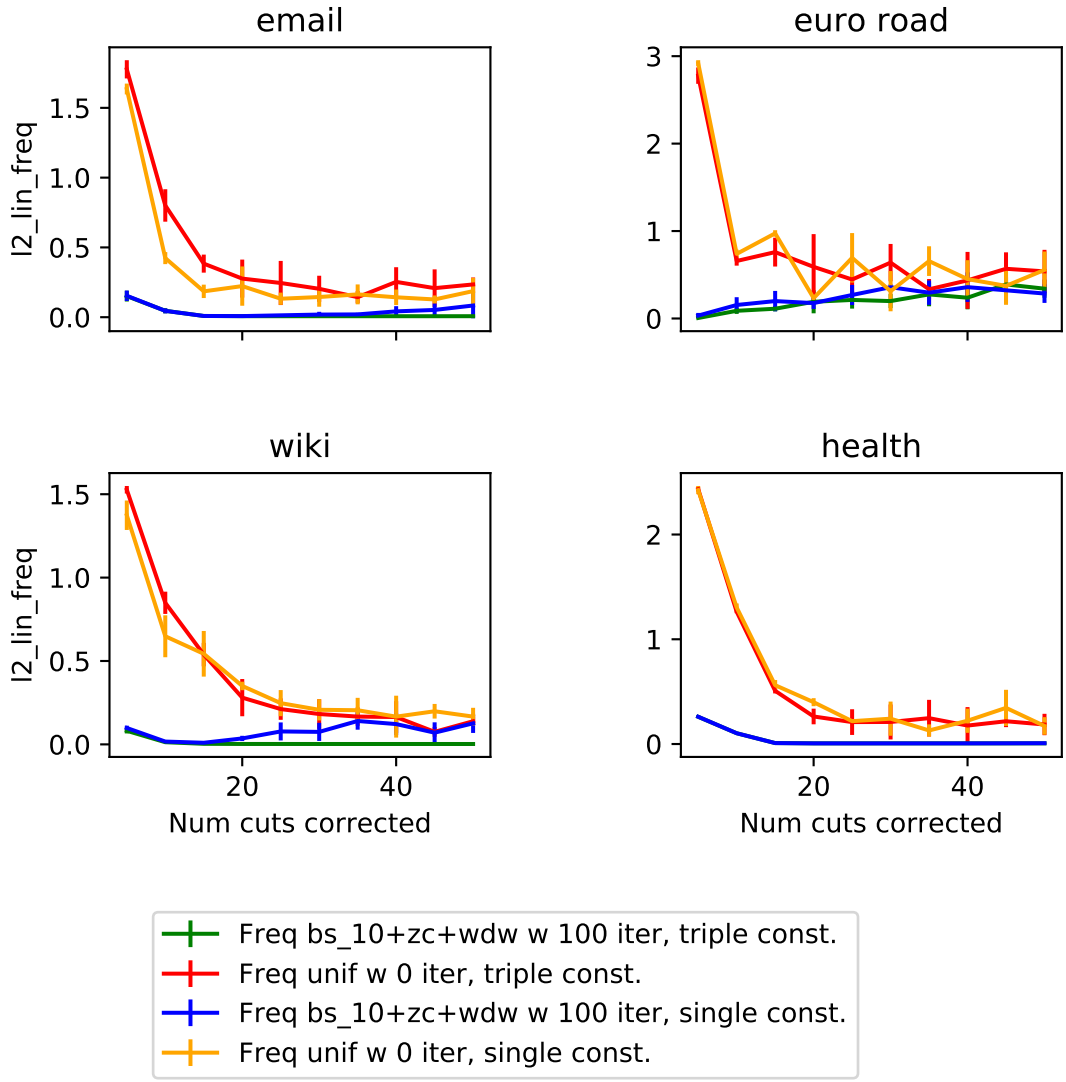


Figure 10.2: Number of cuts corrected against ℓ_2^{LW} ($\boldsymbol{\lambda}^*$,$\boldsymbol{\lambda}$) for Email, EuroRoad, Wiki, and Health graphs.

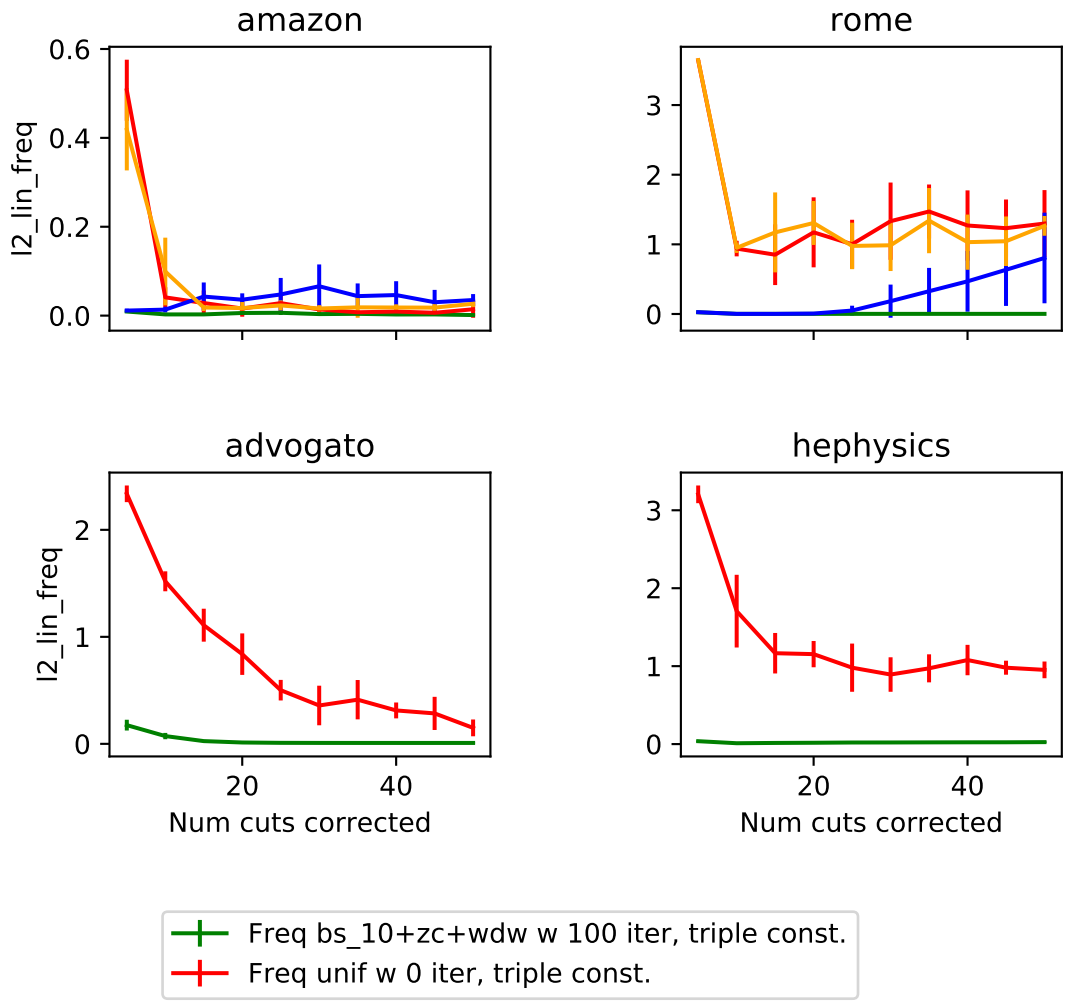


Figure 10.3: Number of cuts corrected against ℓ_2^{LW} $(\boldsymbol{\lambda}^*,\boldsymbol{\lambda})$ for Amazon, Rome, Advogato, and HEPhysics graphs.

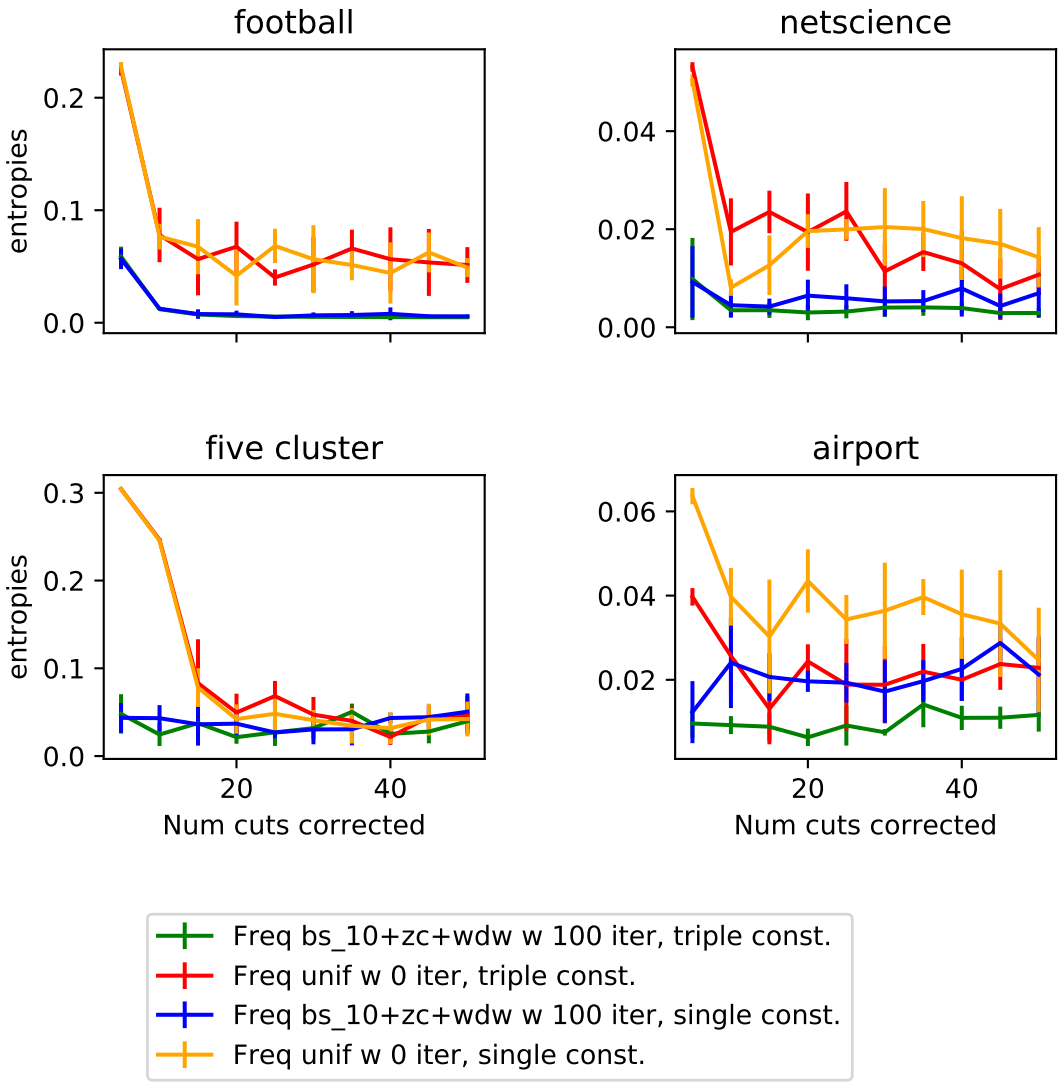


Figure 10.4: Number of cuts corrected against mean entropy $h(a_{u,v})$ for Football, Netscience, Five Cluster, and Airport graphs.

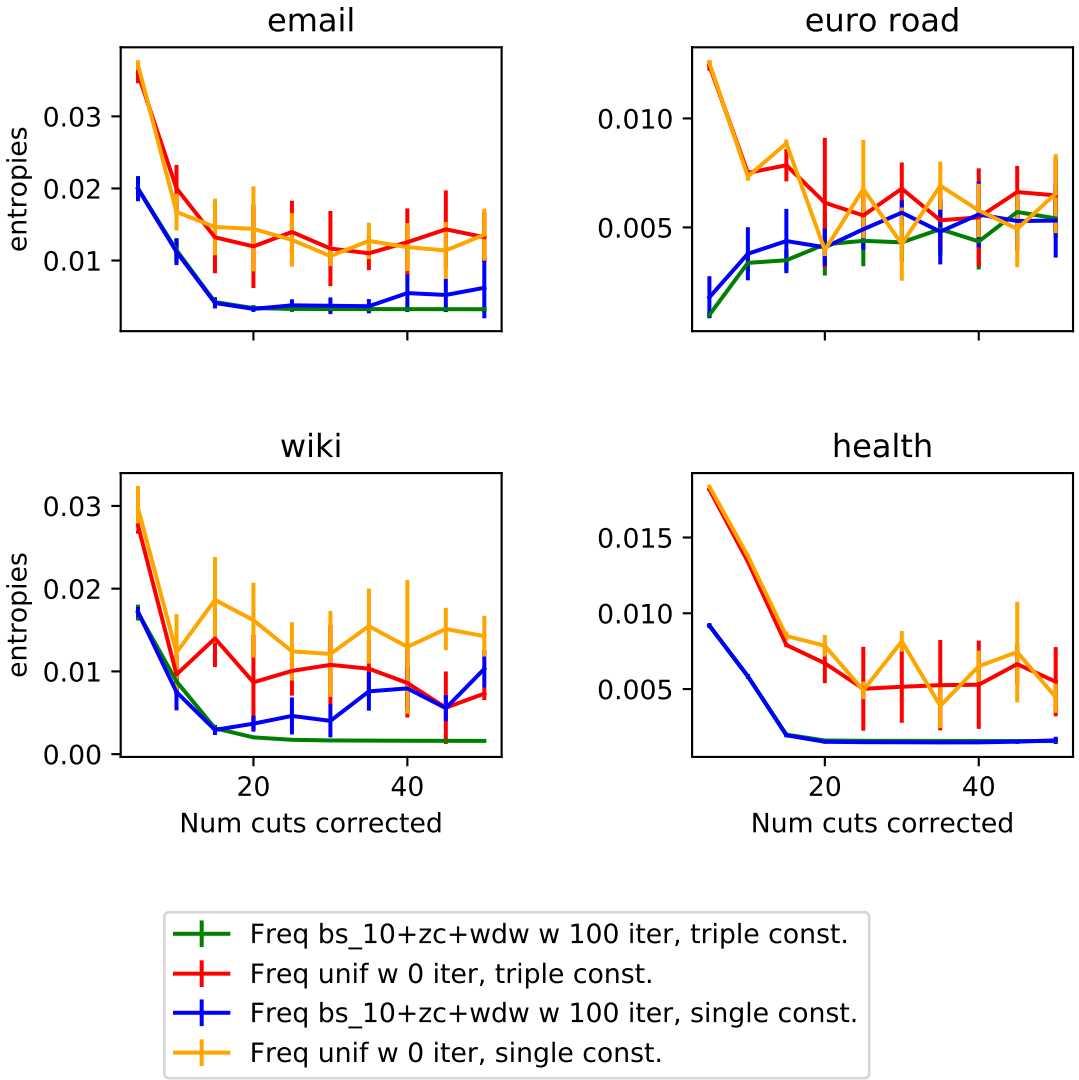


Figure 10.5: Number of cuts corrected against mean entropy $h(a_{u,v})$ for Email, EuroRoad, Wiki, and Health graphs.

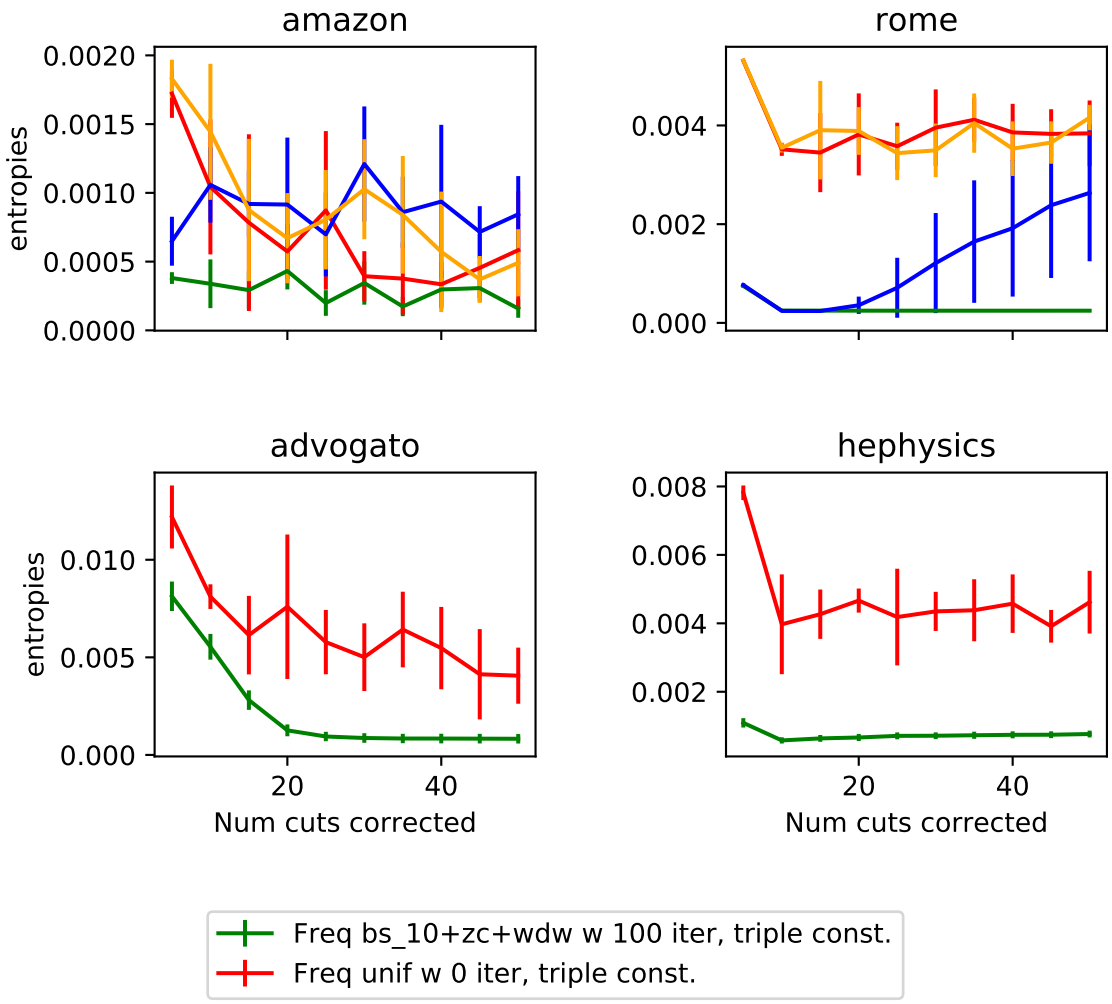


Figure 10.6: Number of cuts corrected against mean entropy $h(a_{u,v})$ for Amazon, Rome, Advogato, and HEPhysics graphs.

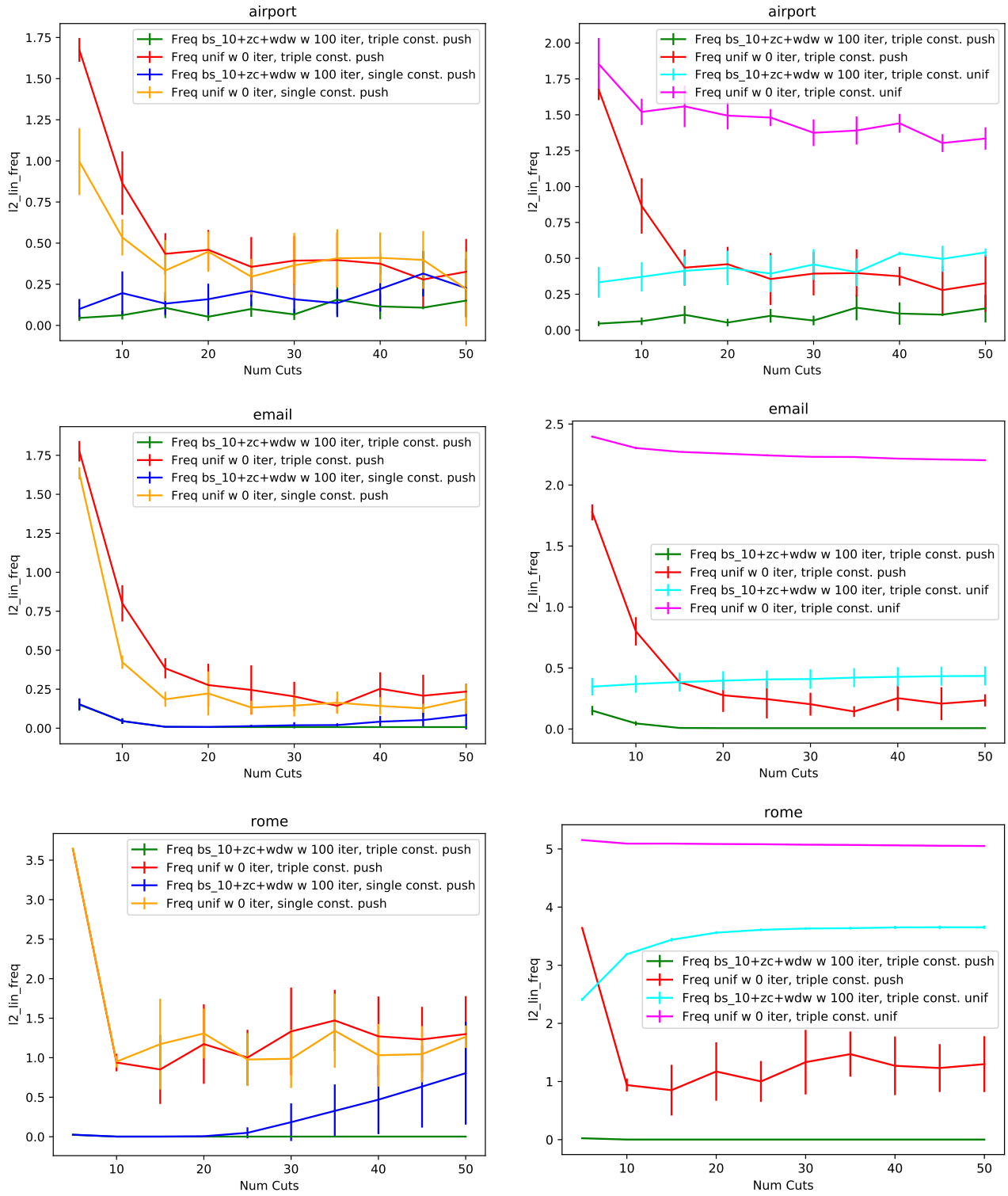


Figure 10.7: Comparing the spectral performance of the different constraints (single and triple) and different updates (uniform and push). The triple constraint seems to be more stable than single on the Random walk-based generation seed probababilistic adjacency matrices, while the push updates improve performance far faster and more reliably than uniform.

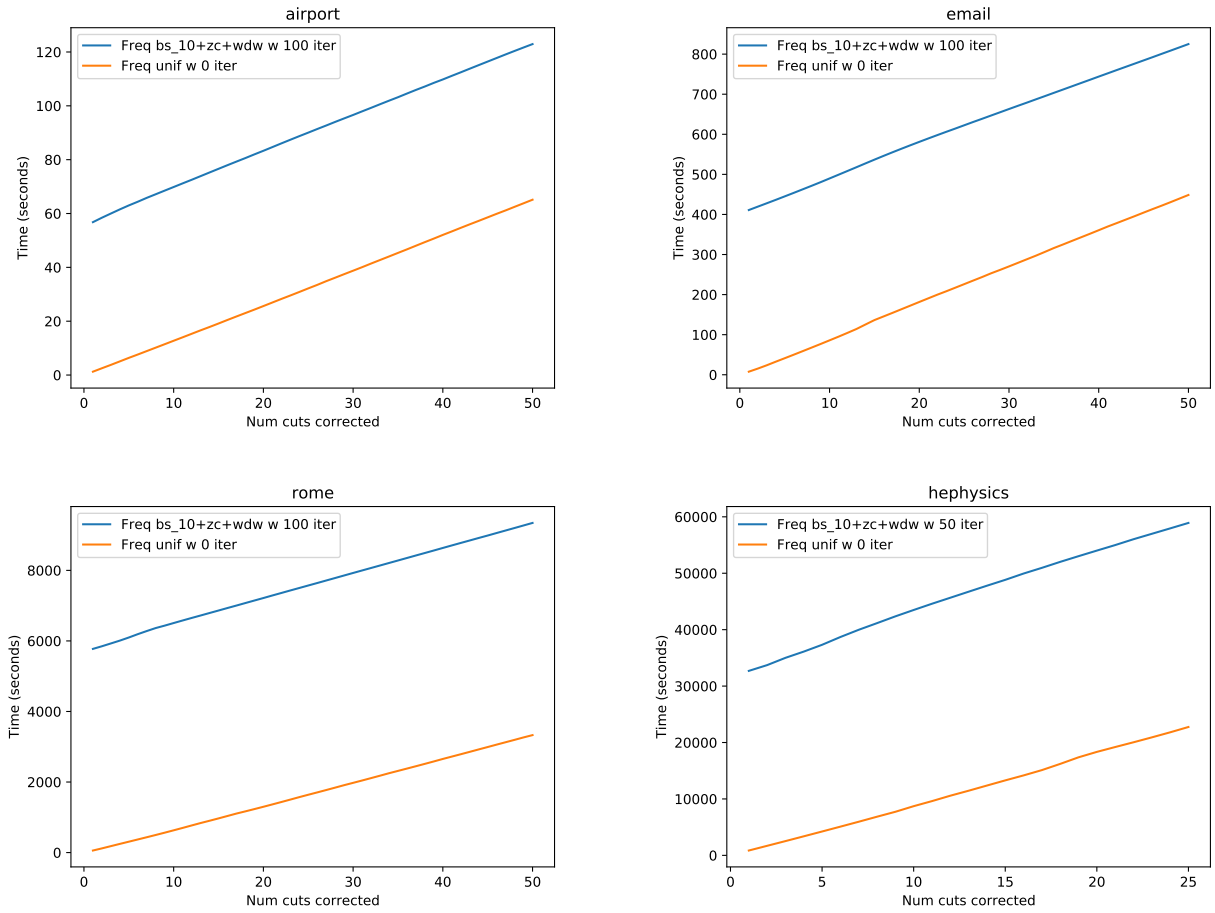


Figure 10.8: Time for correcting cuts in a uniform seed probababilistic adjacency matrix compared with fixing a Random walk-based generation probababilistic adjacency matrix. The time at 0 cuts corrected indicates the time it took to construct the seed matrix. The time it takes to construct the Random walk-based generation probababilistic adjacency matrix is significant compared to the time it takes to correct the cuts.

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 (in seconds) it takes to sample and correct a cut using approximate *GRASP* (Algorithm 26). Means taken from sampling and correcting 50 cuts from a Random walk-based generation seed matrix.

graph HEPhysics (11,204 nodes, 117,619 edges). This is a significant improvement over the NetGAN approach, which does not reach either stopping criterion on the Health graph (2,539 nodes, 5,451 edges) 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 (Table 10.1). We only start to see a decrease in time once graphs reach size approximately 1000. The Airport graph is only 500 nodes and we did not observe an improvement on this input.

Chapter 11

Experimental results comparing models

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 with respect to how well they achieve both the diversity objective (measured using entropy of Bernoulli edge probabilities) and the similarity objective (measured by comparing graph statistics of generated graphs to target graphs). (For further discussion of the similarity and diversity objective, see Chapter 3). The models and hyperparamaters used, along with abbreviations, are listed in Table 11.1.

The graphs are some of the smaller ones presented in Chapter 6 in order to compare against NetGAN for which training takes days once the graphs get to size 2,500 (as seen on the HEALTH graph). For Spectrum-matching generation, we do have additional experiments for graphs up to size approximately 2,500 (the largest graph used was HEALTH) compared to the Kronecker and Stochastic Block models (Shine and Kempe, 2019). Spectrum-matching generation performance and how it compares to the Kronecker and Stochastic Block models for the larger graph inputs is comparable to what is presented in this chapter. For Cut fix generation, results for larger inputs up to size approximately 10,000 (the largest graph used was HEPHYSICS) on the probababilistic adjacency matrices are in Section 10.4.

For the Cut fix generation models, we chose the number of cuts to fix based on where $\ell_2^{\text{LW}}(\boldsymbol{\lambda}(A), \boldsymbol{\lambda}^*)$ stops improving (Figures 10.1, 10.2 and 10.3). After 20 cut corrections, generally the uniform seed stops improving on $\ell_2^{\text{LW}}(\boldsymbol{\lambda}(A), \boldsymbol{\lambda}^*)$. We do consider 50 cuts to see whether or not the performance improves relative to the other models for statistics on the discrete graphs. We correct 50 cuts to investigate if the uniform seed can eventually compete with Random walk-based generation seed. For Random walk-based generation seeds, we only consider graphs after 5 cuts have

Model abbreviation	Model
Walk/0	Random walk-based generation
Walk/5	Random walk-based generation with 5 cuts corrected using Cut fix generation.
Unif/10	Cut fix generation on the *uniform* seed matrix with uniform edge probabilities that sum up to m^* with 10 cuts corrected using Cut fix generation.
Unif/20	Cut fix generation on the *uniform* seed matrix with uniform edge probabilities that sum up to m^* with 20 cuts corrected using Cut fix generation.
Unif/50	Cut fix generation on the *uniform* seed matrix with uniform edge probabilities that sum up to m^* with 50 cuts corrected using Cut fix generation.
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 prediction 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	Spectrum-matching generation
Config	Configuration graph model
Kron	Kronecker with 2×2 seed matrix.
SBM	Stochastic Block Model with 2, 5, 10, and 20 clusters.

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

been corrected because the entropy is so small compared to the other global generation models and the benchmarks (Section 11.1). For all Cut fix generation models, we use the push update in our probabilistic adjacency matrix construction because it was more stable in matching the spectrum as we corrected more cuts in our preliminary experiments (Figure 10.7). We use the single constraint because it has slightly better expected spectral performance for the uniform seed matrix without hurting the spectral performance on the Random walk-based generation results for small numbers of cuts corrected (5 cuts corrected) (Figures 10.1, 10.2 and 10.3).

11.1 Diversity: comparing entropy across models

For most of the models in Table 11.1, the model prescribes for each edge an inclusion probability for independent edge sampling. For these, we measure diversity by computing the average entropy (Section 3.2) of these edge probabilities (Table 11.3). For Spectrum-matching generation, we compute

the entropy after rounding edges down across the critical cuts (Section 7.6).

We find that the global generative models (NetGAN, Random walk-based generation, Cut fix generation with uniform seed) that are terminated earlier have comparable entropy to the Kronecker model. Spectrum-matching generation has low entropy compared to the benchmarks and the global generative models that are terminated earlier. For the Stochastic Block model, the entropy is large compared to all models even once the number of clusters reaches 20.

Graph	Walk/0	Walk/5	Unif/10	Unif/20	Unif/50	NG/S/EP	NG/S/SE	NG/C/EP	NG/C/SE	SpecG
Football	.158 ± .01	.053 ± .01	.089 ± .02	.048 ± .02	.047 ± .02	.169 ± .02	.132 ± .04	**.224 ± .02**	.041 ± .0	0.09 ± 0.01
NetSci	**.022 ± .0**	.004 ± .0	.008 ± .0	.017 ± .01	.018 ± .01	.02 ± .01	.012 ± .0	.019 ± .01	.014 ± .0	0.01 ± 0.0
Airport	.043 ± .0	.015 ± .01	.037 ± .01	.037 ± .01	.027 ± .01	.037 ± .0	.028 ± .01	**.068 ± .01**	.055 ± .01	0.02 ± 0.0
5 Cluster	.113 ± .01	.036 ± .02	**.246 ± .0**	.055 ± .02	.041 ± .02	.14 ± .01	.128 ± .0	.146 ± .02	.131 ± .0	0.02 ± 0.0
Email	.025 ± .0	.02 ± .0	.017 ± .0	.013 ± .0	.014 ± .01	.03 ± .0	.023 ± .01	**.033 ± .0**	.029 ± .0	0.01 ± 0.0

Table 11.2: Mean entropy of entries in the probababilistic adjacency matrices for each model. The walk algorithm combined with correcting cuts results has low entropy compared with NetGAN (Email is an exception).

Graph	Kron	SBM2	SBM5	SBM10	SBM20	Unif
Football	0.268	**0.276**	0.223	0.167	0.131	0.482
NetSci	0.04	**0.06**	0.05	0.04	0.04	0.12
Airport	0.073	**0.107**	0.105	0.103	0.1	0.192
5 Cluster	**0.333**	0.25	0.138	0.136	0.134	0.498
Email	0.023	**0.048**	0.044	0.043	0.041	0.086

Table 11.3: The entropy of the Stochastic Block models with as many as 20 clusters is higher than the global generative graph models and in general higher than the Kronecker model (Five Cluster is an exception).

11.2 Results for different hyperparamaters

We perform a comparison across hyperparamaters for Random walk-based generation, Cut fix generation and NetGAN and display the results in Tables 11.4, 11.5, 11.6, 11.7, 11.8, and 11.9. We find that Cut fix generation combined with Random walk-based generation and NetGAN have better results than Cut fix generation on the uniform seed.

As we saw in Section 8.7.2, adding the additional spectral stopping criterion to the edge prediction stopping criterion in NetGAN helps match graph properties compared to the target. We include the results for both the standard and combination walks using the additional spectral stopping criterion here to compare to the Random walk-based generation and Cut fix generation models.

Investigating the importance of the right seed matrix, notice that Cut fix generation seeded with a Random walk-based generation seed matrix even after only 5 cuts corrected outperforms Cut fix generation with 50 cuts corrected with a uniform seed matrix. In fact, Random walk-based generation tends to do better on similarity even without correcting any cuts compared to Cut fix generation with a uniform seed matrix (up to 50 cuts corrected). In general, we observe that correcting additional cuts helps match properties of $\mathcal{G}^*$ in Cut fix generation for both the uniform and Random walk-based generation seed matrices. However, the improvements stop for the uniform seed matrix in general. To see this, correcting 50 cuts instead of 20 cuts rarely made a difference.

For matching the spectrum measured by ℓ_2^{LW}, NetGAN with the combination walk and additional spectral stopping criterion performs the best on the Five Cluster graph (Table 11.4). 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 aid in matching the spectrum with the Random walk-based generation seed matrix on the Five Cluster graph, suggesting that this method does not work well in the presence of extremely sparse cuts as exist in the Five Cluster graph. This is likely because the Random walk-based 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. Adding the mostly homogeneous updates destroys more structure learned by Random walk-based generation then the structure gained by correcting cuts.

The method that had the largest connected component varied (Table 11.9). 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-based generation seed matrices helped attach nodes because cuts that are too sparse in the seed probababilistic adjacency matrix are corrected and the nodes are reattached. However, for the Five Cluster graph, we see that correcting cuts breaks off nodes. This is likely because the *GRASP* optimization finds a cluster that needs addition noise, but by applying subtraction noise on either side of the cut, it breaks off nodes.

Graph	Walk/0	Walk/5	Unif/20	Unif/50	NG/S/SE	NG/C/SE
Football	0.02 ± 0.0	**0.0 ± 0.0**	0.03 ± 0.01	0.05 ± 0.01	0.01 ± 0.0	0.01 ± 0.0
NetSci	0.03 ± 0.01	**0.0 ± 0.0**	0.07 ± 0.02	0.07 ± 0.02	0.02 ± 0.01	0.01 ± 0.0
5 Cluster	0.05 ± 0.03	0.06 ± 0.08	0.05 ± 0.03	0.05 ± 0.01	0.11 ± 0.14	**0.01 ± 0.01**
Airport	0.1 ± 0.01	**0.07 ± 0.03**	0.12 ± 0.05	0.11 ± 0.07	0.09 ± 0.05	0.1 ± 0.06
Email	0.03 ± 0.01	**0.02 ± 0.0**	0.03 ± 0.01	0.04 ± 0.02	**0.02 ± 0.02**	**0.02 ± 0.01**

Table 11.4: Average difference in spectrum measured by ℓ_2^{LW} $(\boldsymbol{\lambda}^*, \boldsymbol{\lambda})$ between target graphs and random graphs. The comparison is across (1) two seed probabilistic adjacency matrices and different numbers of cuts corrected and (2) NetGAN with spectral edge prediction stopping criterion for both the standard and combination walks.

Graph	Walk/0	Walk/5	Unif/20	Unif/50	NG/S/SE	NG/C/SE
Football	0.06 ± 0.01	**0.01 ± 0.0**	0.1 ± 0.03	0.13 ± 0.03	0.02 ± 0.01	0.04 ± 0.01
NetSci	**0.0 ± 0.0**	**0.0 ± 0.0**	0.06 ± 0.02	0.07 ± 0.03	0.01 ± 0.0	0.01 ± 0.0
5 Cluster	**0.0 ± 0.0**	0.02 ± 0.04	0.08 ± 0.05	0.14 ± 0.03	**0.0 ± 0.01**	**0.0 ± 0.0**
Airport	**0.05 ± 0.01**	0.06 ± 0.03	0.1 ± 0.06	0.07 ± 0.06	**0.05 ± 0.05**	0.08 ± 0.05
Email	0.09 ± 0.01	0.07 ± 0.01	0.06 ± 0.03	0.07 ± 0.04	**0.03 ± 0.02**	0.04 ± 0.02

Table 11.5: Average difference in spectral gap measured by $|\lambda_2^* - \lambda_2|$ between target graphs and random graphs. The comparison is across (1) two seed probabilistic adjacency matrices and different numbers of cuts corrected and (2) NetGAN with spectral edge prediction stopping criterion for both the standard and combination walks.

11.3 Discrete benchmark comparison

We compare the global feature methods to three benchmarks (1) the Configuration model (Config) (2) the Stochastic Block model (SBM) with 5, 10 and 20 clusters and (3) the Kronecker model (Kron). We choose the following hyperparamaters for our global feature methods using the entropy of each model (entropies for each model are shown in Table 11.3):

1. For Random walk-based generation, we compare without cut corrections because once cuts are

Graph	Walk/0	Walk/5	Unif/20	Unif/50	NG/S/SE	NG/C/SE
Football	0.1 ± 0.01	**0.04 ± 0.01**	0.11 ± 0.03	0.14 ± 0.02	**0.04 ± 0.01**	0.05 ± 0.01
NetSci	0.5 ± 0.11	**0.4 ± 0.21**	1.61 ± 0.28	1.77 ± 0.17	1.26 ± 0.17	1.05 ± 0.3
5 Cluster	**1.2 ± 0.16**	2.24 ± 0.84	3.28 ± 0.16	3.5 ± 0.05	2.02 ± 0.35	2.5 ± 0.37
Airport	0.26 ± 0.01	0.14 ± 0.03	0.14 ± 0.05	0.2 ± 0.1	0.18 ± 0.07	**0.12 ± 0.04**
Email	0.22 ± 0.02	0.2 ± 0.02	0.15 ± 0.06	0.15 ± 0.05	0.2 ± 0.07	**0.13 ± 0.05**

Table 11.6: Average difference in shortest path length distribution measured by Earth mover's distance between target graphs and random graphs. The comparison is across (1) two seed probabilistic adjacency matrices and different numbers of cuts corrected and (2) NetGAN with spectral edge prediction stopping criterion for both the standard and combination walks.

Graph	Walk/0	Walk/5	Unif/20	Unif/50	NG/S/SE	NG/C/SE
Football	0.13 ± 0.01	**0.03 ± 0.0**	0.17 ± 0.02	0.18 ± 0.03	0.06 ± 0.01	0.08 ± 0.02
NetSci	0.37 ± 0.01	**0.11 ± 0.08**	0.63 ± 0.04	0.59 ± 0.06	0.25 ± 0.02	0.23 ± 0.04
5 Cluster	0.13 ± 0.05	0.08 ± 0.04	0.17 ± 0.04	0.17 ± 0.02	0.05 ± 0.01	**0.04 ± 0.01**
Airport	0.21 ± 0.01	0.27 ± 0.09	0.47 ± 0.06	0.44 ± 0.09	**0.17 ± 0.04**	0.34 ± 0.09
Email	0.09 ± 0.01	0.08 ± 0.01	0.19 ± 0.01	0.18 ± 0.02	**0.06 ± 0.05**	0.11 ± 0.04

Table 11.7: Average difference in clustering coefficient distribution measured by Earth mover's distance between target graphs and random graphs. The comparison is across (1) two seed probabilistic adjacency matrices and different numbers of cuts corrected and (2) NetGAN with spectral edge prediction stopping criterion for both the standard and combination walks.

Graph	Walk/0	Walk/5	Unif/20	Unif/50	NG/S/SE	NG/C/SE
Football	1.69 ± 0.11	**0.97 ± 0.14**	2.41 ± 0.3	2.55 ± 0.39	1.66 ± 0.19	1.06 ± 0.07
NetSci	0.56 ± 0.09	**0.2 ± 0.06**	0.79 ± 0.11	0.71 ± 0.12	0.27 ± 0.08	0.49 ± 0.12
5 Cluster	11.0 ± 4.15	5.57 ± 1.75	9.1 ± 2.71	**4.52 ± 0.64**	5.94 ± 0.55	5.47 ± 0.56
Airport	**3.48 ± 0.28**	3.51 ± 0.87	6.41 ± 0.75	6.7 ± 0.96	1.15 ± 0.38	4.2 ± 1.05
Email	0.66 ± 0.09	**0.42 ± 0.07**	1.67 ± 0.52	2.4 ± 0.51	0.65 ± 0.24	1.84 ± 0.18

Table 11.8: Average difference in degree distribution, measured by Earth mover's distance, between target graphs and random graphs generated. The comparison is across (1) two seed probabilistic adjacency matrices and different numbers of cuts corrected and (2) NetGAN with spectral edge prediction stopping criterion for both the standard and combination walks.

Graph	Walk/0	Walk/5	Unif/10	Unif/20	Unif/50	NG/S/SE	NG/C/SE
Football	115 ± 0	115 ± 0	115 ± 0	115 ± 0	115 ± 0	115 ± 0	115 ± 0
NetSci	351 ± 8	**373 ± 3**	371 ± 3	360 ± 16	364 ± 9	367 ± 9	362 ± 9
5 Cluster	463 ± 26	462 ± 53	**500 ± 0**	**500 ± 0**	**500 ± 0**	428 ± 83	497 ± 7
Airport	495 ± 2	**496 ± 4**	446 ± 61	466 ± 36	465 ± 33	476 ± 18	479 ± 11
Email	1111 ± 5	1110 ± 6	1068 ± 20	1104 ± 27	**1122 ± 9**	1068 ± 26	1117 ± 6

Table 11.9: Average Size of the largest connected component of random graphs. The comparison is across (1) two seed probabilistic adjacency matrices and different numbers of cuts corrected and (2) NetGAN with spectral edge prediction stopping criterion for both the standard and combination walks.

corrected the entropies are much lower than the Stochastic Block model and Kronecker model. Because we see that lower-entropy models generally perform better on our graph similarity metrics, to make the comparison more fair, we use Walk/0.

2. For Cut fix generation, we compare using the uniform seed matrix and 10 cut corrections. We saw above that the uniform seed matrix lags behind the Random walk-based generation seed for most graph similarity metrics even as the number of cut corrections reaches 50. However, we are interested in whether Cut fix generation with the uniform seed can compete against the benchmarks that 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 both the other global methods and the Kronecker model. Additionally, the similarity performance improves compared to using the edge prediction stopping criterion only.

Box and whisker plots showing the randomness over each model and randomized rounding are in Section 11.3.1. Tables with means and standard deviations are in Section 11.3.2.

For all graph similarity metrics besides degree distribution and the size of the largest connected component, the global feature method that did best on each input did better (spectrum ℓ_2^{LW}, spectral gap absolute value, shortest path) or the same (clustering coefficient and betweenness) than the benchmarks. NetGAN tends to do the best, with the exception of the spectral gap for which Spectrum-matching generation is superior (Figure 11.2). Spectrum-matching generation likely matches the spectral gap better than the other approaches due to the aggressive rounding across the critical cuts in Algorithm 2. As we add more clusters, the Stochastic Block model starts to catch up for most graphs and features. Once the number of cluster reaches 20, the Stochastic Block model is typically just as good as the best global model but for some inputs still seems to have trouble (most noticeably, NETSCIENCE).

Like NetGAN with the standard random walk, Spectrum-matching generation is susceptible to disconnecting the FIVE CLUSTER graph so it lagged behind the Random walk-based generation for matching the spectral gap after breaking off clusters (Figure 11.7). For the size of the largest connected component, the Configuration and Stochastic Block models perform best. The Kronecker

model noticeably lags behind because its probabilistic graphs are limited to sizes that are powers of 2.

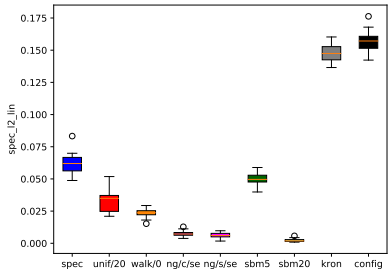
11.3.1 Plots

The plots in this section are all box and whisker plots generated using Matplotlib (Hunter, 2007). For each plot, the box extends from the lower to upper quartile values ($Q1$ and $Q3$) of the data with a line at the median. The whiskers are computed from 1.5 times *interquartile range* (IQR) which is the difference $\text{IQR} = Q3 - Q1$. The lower whisker extends from the end of the box at $Q1$ to $Q1 - 1.5\text{IQR}$ and the upper whisker extends from the top of the box at $Q3$ to $Q3 + 1.5\text{IQR}$. Any additional data points that extend beyond the whiskers are outliers and are marked by circles.

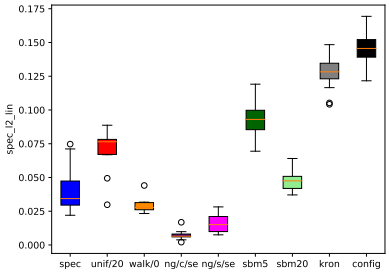
The randomness is over 40 random graphs drawn for each of the global generative models except for Spectrum-matching generation where we draw one random graph for each model. For the benchmarks, we draw 40 random graphs for each one. We train 10 models for each of the global generative models.

11.3.2 Tables

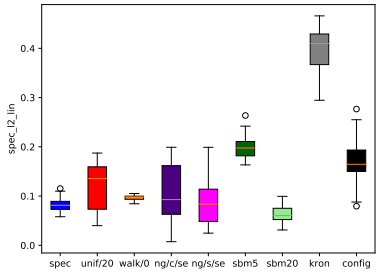
FOOTBALL



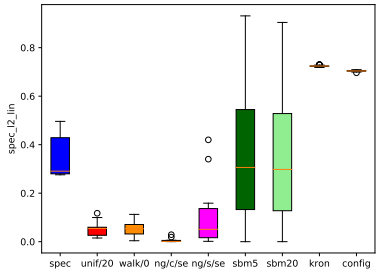
NETSCIENCE



AIRPORT



FIVE CLUSTER



EMAIL

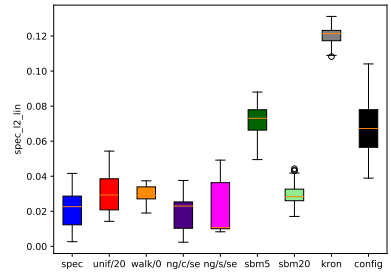
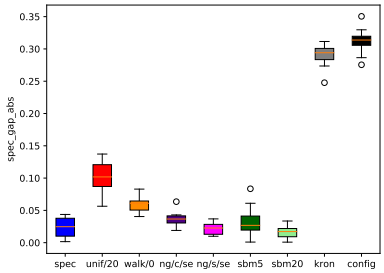
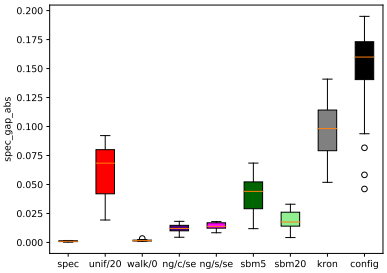


Figure 11.1: Box and whisker plots for the difference in spectrum, measured by $\ell_2^{\mathrm{LW}}(\boldsymbol{\lambda}^*, \boldsymbol{\lambda})$, between target graphs and random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

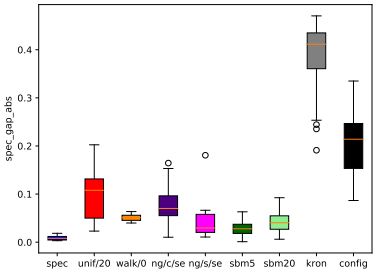
FOOTBALL



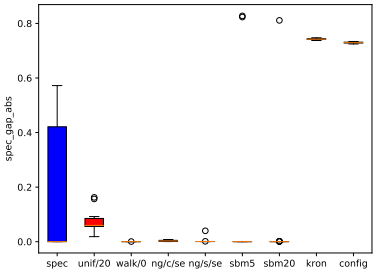
NETSCIENCE



AIRPORT



FIVE CLUSTER



EMAIL

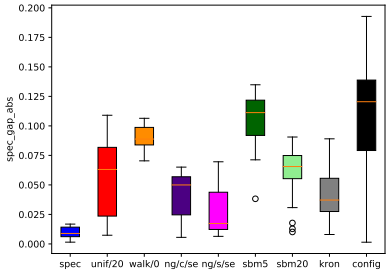
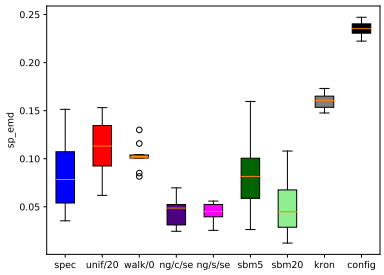
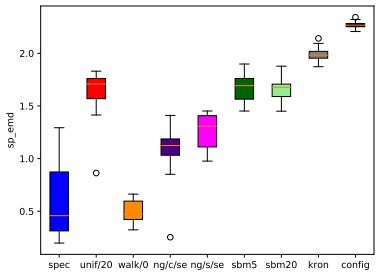


Figure 11.2: Box and whisker plots for the difference in spectral gap, measured by $|\lambda_2^* - \lambda_2|$, between target graphs and random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

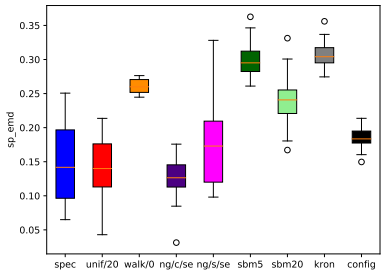
Football



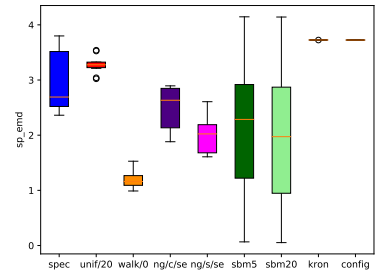
Netscience



Airport



Five Cluster



Email

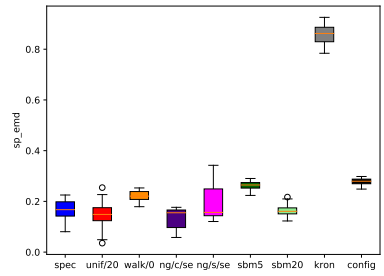
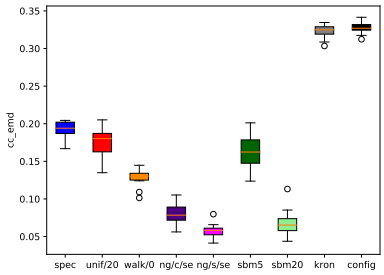
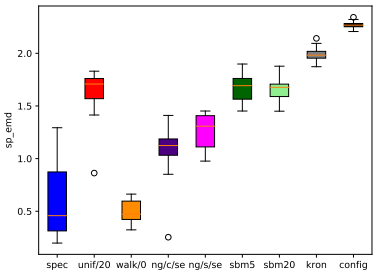


Figure 11.3: Box and whisker plots for the difference in shortest path length distributions, measured by Earth mover's distance, between target graphs and random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

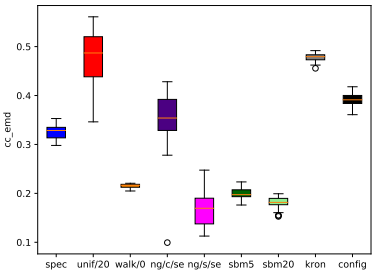
Football



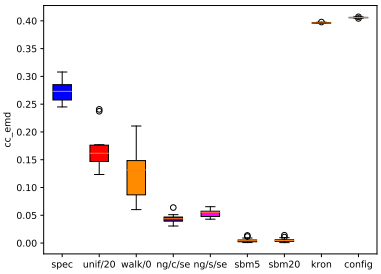
Netscience



Airport



Five Cluster



Email

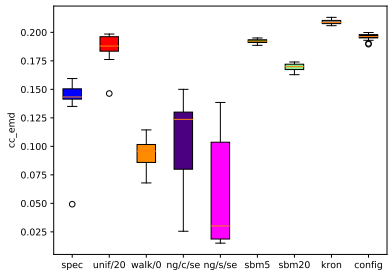
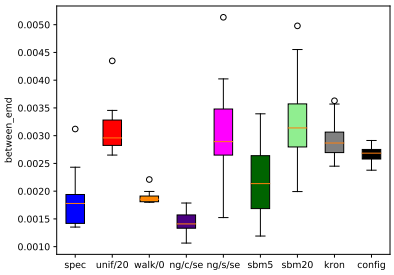
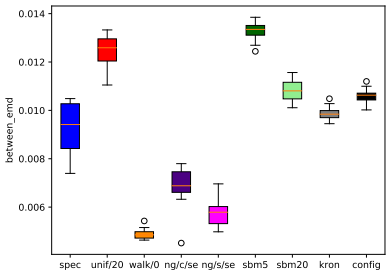


Figure 11.4: Box and whisker plots for the difference in clustering coefficient distributions, measured by Earth mover's distance, between target graphs and random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

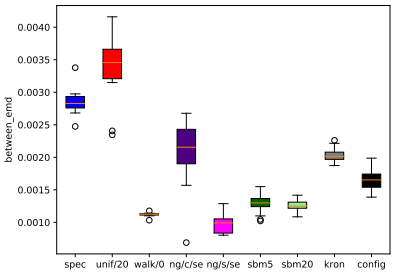
Football



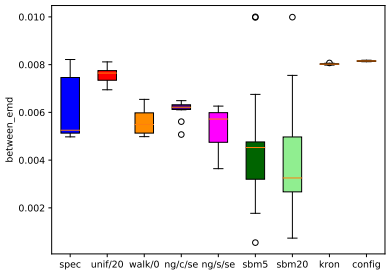
Netscience



Airport



Five Cluster



Email

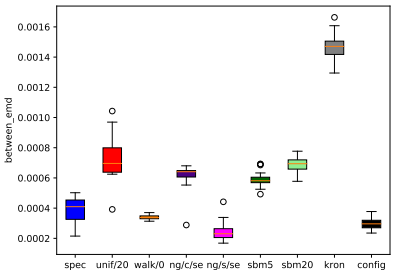
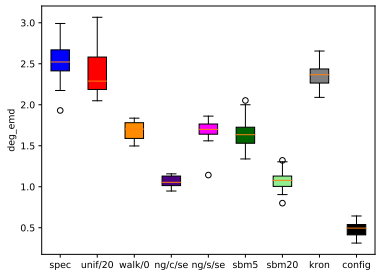
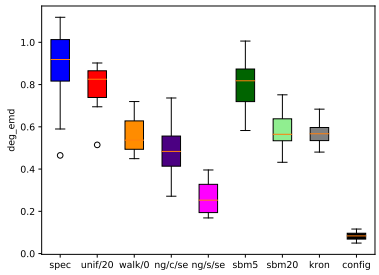


Figure 11.5: Box and whisker plots for the difference in betweenness centrality distributions, measured by Earth mover's distance, between target graphs and random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

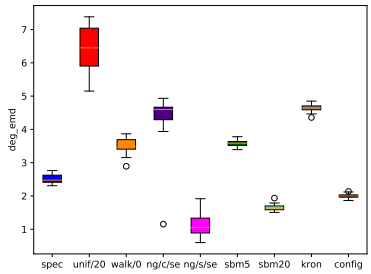
Football



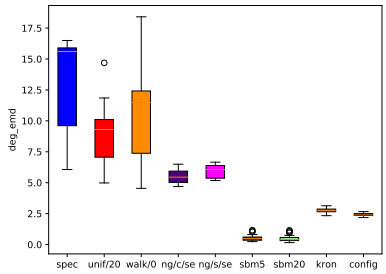
Netscience



Airport



Five Cluster



Email

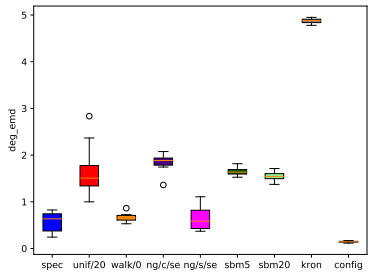
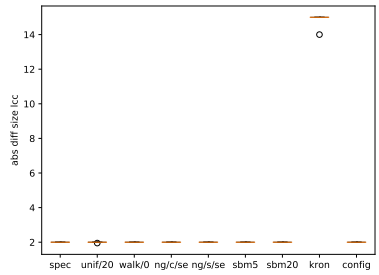
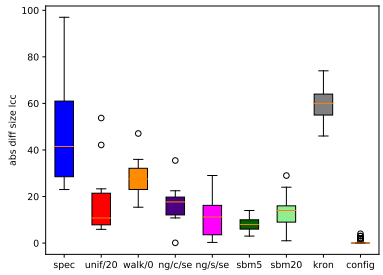


Figure 11.6: Box and whisker plots for the difference in degree distributions, measured by Earth mover's distance, between target graphs and random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

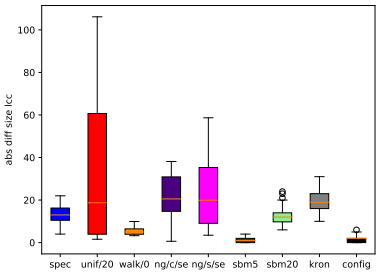
FOOTBALL



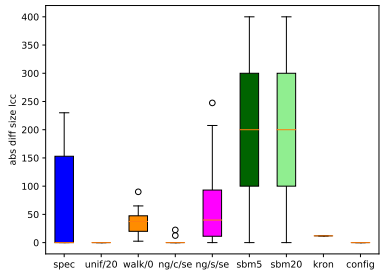
NETSCIENCE



AIRPORT



FIVE CLUSTER



EMAIL

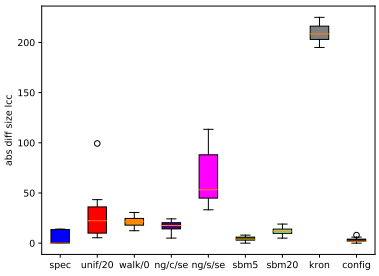


Figure 11.7: Box and whisker plots for absolute value of the difference in size of the largest connected component between target graphs and random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

Graph	Walk/0	Unif/20	NG/S/SE	NG/C/SE	SpecG	Config	SBM5	Kron
Football	0.02 ± 0.0	0.03 ± 0.01	**0.01 ± 0.0**	0.01 ± 0.0	0.06 ± 0.01	0.16 ± 0.01	0.05 ± 0.0	0.15 ± 0.01
NetSci	0.03 ± 0.01	0.07 ± 0.02	0.02 ± 0.01	**0.01 ± 0.0**	0.04 ± 0.02	0.15 ± 0.01	0.09 ± 0.01	0.13 ± 0.01
Airport	0.1 ± 0.01	0.12 ± 0.05	0.09 ± 0.05	0.1 ± 0.06	**0.08 ± 0.02**	0.17 ± 0.04	0.2 ± 0.02	0.4 ± 0.04
5 Cluster	0.05 ± 0.03	0.05 ± 0.03	0.11 ± 0.14	**0.01 ± 0.01**	0.34 ± 0.09	0.7 ± 0.0	0.34 ± 0.25	0.72 ± 0.0
Email	0.03 ± 0.01	0.03 ± 0.01	**0.02 ± 0.02**	**0.02 ± 0.01**	**0.02 ± 0.01**	0.07 ± 0.02	0.07 ± 0.01	0.12 ± 0.01

Table 11.10: Average difference in spectrum, measured by ℓ_2^{LW} $(\boldsymbol{\lambda}^*, \boldsymbol{\lambda})$, between target graphs and random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

Graph	Walk/0	Unif/20	NG/S/SE	NG/C/SE	SpecG	Config	SBM5	Kron
Football	0.06 ± 0.01	0.1 ± 0.03	0.02 ± 0.01	0.04 ± 0.01	**0.02 ± 0.01**	0.31 ± 0.01	0.03 ± 0.02	0.29 ± 0.01
NetSci	0.0 ± 0.0	0.06 ± 0.02	0.01 ± 0.0	0.01 ± 0.0	**0.0 ± 0.0**	0.15 ± 0.03	0.04 ± 0.01	0.1 ± 0.02
Airport	0.05 ± 0.01	0.1 ± 0.06	0.05 ± 0.05	0.08 ± 0.05	**0.01 ± 0.0**	0.21 ± 0.06	0.03 ± 0.01	0.38 ± 0.08
5 Cluster	**0.0 ± 0.0**	0.08 ± 0.05	**0.0 ± 0.01**	0.0 ± 0.0	0.17 ± 0.26	0.73 ± 0.0	0.04 ± 0.18	0.74 ± 0.0
Email	0.09 ± 0.01	0.06 ± 0.03	0.03 ± 0.02	0.04 ± 0.02	**0.01 ± 0.0**	0.11 ± 0.05	0.11 ± 0.02	0.04 ± 0.02

Table 11.11: Average difference in spectral gap, measured by $|\lambda_2^* - \lambda_2|$, between target graphs and random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

Graph	Walk/0	Unif/20	NG/S/SE	NG/C/SE	SpecG	Config	SBM5	Kron
Football	0.1 ± 0.01	0.11 ± 0.03	**0.04 ± 0.01**	0.05 ± 0.01	0.08 ± 0.04	0.24 ± 0.01	0.08 ± 0.03	0.16 ± 0.01
NetSci	0.5 ± 0.11	1.61 ± 0.28	1.26 ± 0.17	1.05 ± 0.3	**0.62 ± 0.4**	2.27 ± 0.03	1.67 ± 0.11	1.99 ± 0.05
Airport	0.26 ± 0.01	**0.14 ± 0.05**	0.18 ± 0.07	0.12 ± 0.04	0.15 ± 0.06	0.18 ± 0.01	0.3 ± 0.03	0.31 ± 0.02
5 Cluster	**1.2 ± 0.16**	3.28 ± 0.16	2.02 ± 0.35	2.5 ± 0.37	2.93 ± 0.57	3.72 ± 0.0	2.1 ± 1.05	3.73 ± 0.0
Email	0.22 ± 0.02	0.15 ± 0.06	0.2 ± 0.07	**0.13 ± 0.05**	0.17 ± 0.04	0.28 ± 0.01	0.26 ± 0.02	0.86 ± 0.04

Table 11.12: Average difference in shortest path length distribution, measured by Earth mover's distance, between target graphs and random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

Graph	Walk/0	Unif/20	NG/S/SE	NG/C/SE	SpecG	Config	SBM5	Kron
Football	0.13 ± 0.01	0.17 ± 0.02	**0.06 ± 0.01**	0.08 ± 0.02	0.19 ± 0.01	0.33 ± 0.01	0.16 ± 0.02	0.32 ± 0.01
NetSci	0.37 ± 0.01	0.63 ± 0.04	0.25 ± 0.02	**0.23 ± 0.04**	0.49 ± 0.03	0.72 ± 0.01	0.69 ± 0.01	0.72 ± 0.0
Airport	0.21 ± 0.01	0.47 ± 0.06	**0.17 ± 0.04**	0.34 ± 0.09	0.33 ± 0.02	0.39 ± 0.01	0.2 ± 0.01	0.48 ± 0.01
5 Cluster	0.13 ± 0.05	0.17 ± 0.04	0.05 ± 0.01	0.04 ± 0.01	0.27 ± 0.02	0.41 ± 0.0	**0.0 ± 0.0**	0.4 ± 0.0
Email	0.09 ± 0.01	0.19 ± 0.01	**0.06 ± 0.05**	0.11 ± 0.04	0.14 ± 0.03	0.2 ± 0.0	0.19 ± 0.0	0.21 ± 0.0

Table 11.13: Average difference in clustering coefficient distribution, measured by Earth mover's distance, between target graphs and random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

Graph	Walk/0	Unif/20	NG/S/SE	NG/C/SE	SpecG	Config	SBM5	Kron
Football	1.69 ± 0.11	2.41 ± 0.3	1.66 ± 0.19	1.06 ± 0.07	2.52 ± 0.29	**0.48 ± 0.08**	1.65 ± 0.15	2.36 ± 0.14
NetSci	0.56 ± 0.09	0.79 ± 0.11	0.27 ± 0.08	0.49 ± 0.12	0.88 ± 0.2	**0.08 ± 0.02**	0.8 ± 0.1	0.57 ± 0.05
Airport	3.48 ± 0.28	6.41 ± 0.75	**1.15 ± 0.38**	4.2 ± 1.05	2.52 ± 0.15	2.0 ± 0.06	3.58 ± 0.09	4.64 ± 0.1
5 Cluster	11.0 ± 4.15	9.1 ± 2.71	5.94 ± 0.55	5.47 ± 0.56	13.03 ± 4.1	2.42 ± 0.11	**0.52 ± 0.21**	2.74 ± 0.15
Email	0.66 ± 0.09	1.67 ± 0.52	0.65 ± 0.24	1.84 ± 0.18	0.57 ± 0.21	**0.14 ± 0.01**	2.05 ± 0.07	4.87 ± 0.04

Table 11.14: Average difference in degree distribution, measured by Earth mover's distance, between target graphs and random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

Graph	Walk/0	Unif/20	NG/S/SE	NG/C/SE	SpecG	Config	SBM5	Kron
Football	**115 ± 0**	**115 ± 0**	**115 ± 0**	**115 ± 0**	**115 ± 0**	**115 ± 0**	**115 ± 0**	128 ± 0
NetSci	351 ± 8	360 ± 16	367 ± 9	362 ± 9	330 ± 25	**379 ± 1**	371 ± 3	439 ± 6
Airport	495 ± 2	466 ± 36	476 ± 18	479 ± 11	487 ± 5	499 ± 2	**499 ± 1**	481 ± 5
5 Cluster	**463 ± 26**	**500 ± 0**	428 ± 83	497 ± 7	436 ± 98	**500 ± 0**	305 ± 122	512 ± 0
Email	1111 ± 5	1104 ± 27	1068 ± 26	1117 ± 6	1128 ± 7	1130 ± 2	**1129 ± 2**	924 ± 8

Table 11.15: Size of the largest connected component of random graphs generated by generative graph models. The comparison is between global generative graph models and benchmark models that have comparable entropy.

Chapter 12

Implementation Details

In this chapter we provide any code bases used as well as the hyper-parameters used to generate all plots and tables. All of our code is online (Shine, 2020).

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 7.6), 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 mainly used the code from (Bojchevski et al., 2018) except we sample the initial vertex using the stationary distribution of the random walk instead of uniformly. To do this, we generalize the

random walk sampling code to use any random walk transition matrix.

12.3 Random walk generation

In our implementation of Algorithm 18, we use a minimum size of 5 for the 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 20 steps apart.

12.4 Cut fix generation

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

In our implementation of Algorithm 27, the parameter 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 configuration model is described 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) for all numbers of clusters.

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 initialize 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 $k = \lceil \log(n^*) \rceil$ using $K^{[k]}$ as a probabilistic adjacency matrix.

Chapter 13

Random graphs using local features and expansion

Chung et al. (2004) show that with high-probability random graphs subject to any degree distribution are indeed *expanders*. Intuitively, the random graphs have high conductance with high probability because the number of ways to wire edges such that they are clustered is small compared to the number of ways to place edges such that they expand (Definition 5.1.3). One way to interpret conductance is the probability that a randomly chosen edge adjacent to a cut will cross the cut. This suggests that if we can bias stub wiring in the Configuration model to encourage clustering, then the 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 question we aim to address in this chapter is: are random graphs drawn uniformly subject to a fixed motif distribution, other than the degree distribution, with high probability expanders? If we fix more complex motif distributions that encourage clustering, are low-conductance graphs drawn more often? Are there more complex motif distributions such that a random graph drawn uniformly subject to these distributions results in a low-conductance graph with high probability?

We ideally would like to (1) show that 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. While we currently lack any general theoretical results relating fixing motifs to conductance, we use a generalization of the Configuration model by Karrer and Newman (2010) to fix motifs up to size 4. We describe Karrer and Newman's model that accommodates arbitrary distributions of sub-graphs in Section 4.1. We observe that fixing motifs

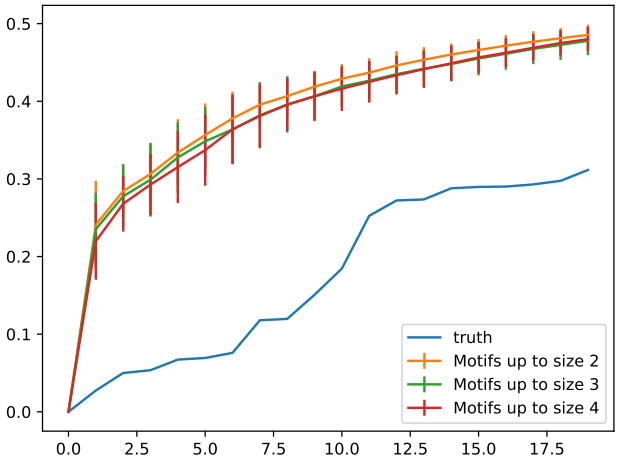


Figure 13.1: Average eigenvalues of the symmetric normalized Laplacian for the AIRPORT graph using the Karrer-Newman Configuration model generalization to generate random graphs subject to distributions of motifs. We construct the distribution over motifs by labeling each edge as part of exactly one motif. Each unlabeled edge is labeled it with the largest motif it is a part of with other unlabeled edges. Edges adjacent to each node are visited in a random order.

up to size 4 is insufficient to prevent generating expanders with high probability (Figure 13.1). However, note that in general fixing motifs of size 4 generates graphs with smaller expansion than fixing degrees (size 2). This suggests that some formulation of our original conjecture might be true: that fixing larger motifs helps preserve low-conductance cuts but the effects are small and fixing local motifs is insufficient to preserve global structure. We discuss further directions in Section 14.2.

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. 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 output graphs with sparse cuts connected on inputs the classic NetGAN can not.

Matching global features is a departure from a long history of generative graph models built around local features or a high-level partition. Matching global features is challenging because (1) these features are computed from the entire graph and (2) of the way they interact with one another. Because the features are computed from the entire graph, each feature can not be localized to one portion of the graph and then matched by building one portion of the graph at a time (as is done with matching local features). Furthermore, the features themselves can not be divided into localized parts (as is the case with a partition). This makes it difficult to match one feature at a time and much of the work in this thesis is accommodating these limitations. For example, our Cut fix generation model sequentially samples cuts and adds/removes edges crossing these cuts in a seed graph so that the number of edges crossing matches the number crossing in the target graph (Chapter 10). It is important to choose cuts and correct them in a way that ensures progress. That is, the cuts corrected to match the target remain close to the target after later steps. For all three of our generative graph models, we introduce intricate heuristics to match these global features that address the complex ways these features interact.

Each of our models is extensively tested to understand how best to approach generating graphs subject to global features. 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 distances and Laplacian spectra, compared to a number of benchmarks. Our generative graph models can compete with the benchmark models not only on the goal of 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 Spectrum-matching generation or NetGAN:

1. Spectrum-matching generation can not scale beyond a couple thousand nodes due to the number of eigendecompositions.

2. NetGAN can not reach either stopping criterion within a day once graphs reach size of a few thousand.

However, many graphs of interest have size on the order of millions of nodes, and as is, Cut fix generation is not able to accommodate such large graphs.

Another direction is exploring simpler heuristics and finding theoretical guarantees for them. The methods presented in this thesis rely on complex heuristics and do not have guarantees. We explore further opportunities for future work below.

14.1 Formalizing the 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; we measure 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 features. 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 for which a model that matches more of these features or matches these features closer must come at an entropy cost? If entropy is not able to capture/express such a trade-off, is there another diversity measure that makes this trade-off explicit?

14.2 Formalizing the trade-off between matching global and local features

The initial motivation for our work was around a very intriguing general direction. We know that generating graphs subject to most degree distributions produces graphs with high conductance, with high probability (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). Unfortunately, without any qualifications, the statement that a random graph drawn uniformly subject to any motif distribution is with high probability an expander is false. One counter example is a 3-regular graph in which each node participates in 3 triangles must be the union of disjoint 4-cliques. This graph is 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. Perhaps fixing local motifs under the conditions that the motif densities are large enough (to ensure enough edges) and sampling over connected graphs (to prevent cases like the one above) is enough to show that with high probability sampling uniformly would result in an expander.

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.

Namrata Anand and Possu Huang. Generative modeling for protein structures. In *Advances in neural information processing systems*, pages 7494–7505. 2018.

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 *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, 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 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 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 Chayes, and Amin Saberi. On the spread of viruses on the internet. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 301–310. SIAM, 2005.

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 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.

Azzedine Boukerche. *Algorithms and protocols for wireless and mobile ad hoc networks*, volume 77. John Wiley & Sons, 2008.

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.

Pierre Buesser, Fabio Daolio, and Marco Tomassini. Optimizing the robustness of scale-free networks with simulated annealing. In *International conference on adaptive and natural computing algorithms*, pages 167–176. Springer, 2011.

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. *International Conference on Knowledge Discovery and Data Mining*, 10(4):49, 2016.

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

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

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

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. 100(11):6313–6318, 2003.

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

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. *International Conference on Machine Learning, ICML 2018 Workshop on Theoretical Foundations and Applications of Deep Generative Models*, 2018.

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

Kayhan Erciyes. *Distributed graph algorithms for computer networks*. Springer Science & Business Media, 2013.

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

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

Matthew Fickus, Dustin Mixon, Miriam 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.

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

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

Michel Goemans and David Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, 42(6): 1115–1145, 1995.

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.

Xiaojie Guo and Liang Zhao. A systematic survey on deep generative models for graph generation, 2020.

Alexander Gutfraind, Ilya Safro, and Lauren 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 Holland, Kathryn Laskey, and Samuel Leinhardt. Stochastic blockmodels: Some first steps. *Social Networks*, 5:109–137, 1983.

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

J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3): 90–95, 2007. doi: 10.1109/MCSE.2007.55.

Matthew 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 Newman. Random graphs containing arbitrary distributions of subgraphs. *Physical Review E*, 82(6):066118, 2010.

Brian Karrer and Mark 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 Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Durk Kingma, Shakir Mohamed, Danilo 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.

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

James Lee, Shayan 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 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.

James MacQueen. Some methods for classification and analysis of multivariate observations. 1967.

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

Alan Mislove, Massimiliano Marcon, Krishna 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 Newman. Finding community structure in networks using the eigenvectors of matrices. 74, 2006.

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

Mark Newman, Steven Strogatz, and Duncan 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 Dankulov, Pol Colomer-de Simón, Almerima Jamakovic, Priya Mahadevan, Amin Vahdat, Kevin Bassler, Zoltán Toroczkai, Marián Boguñá, Guido Caldarelli, Santo Fortunato, and Dmitri Kiroukov. Quantifying randomness in real networks. *Nature Communications*, 6:8627, 2015.

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

Georgios Pavlopoulos, Maria Secrier, Charalampos Moschopoulos, Theodoros Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis Bagos. Using graph theory to analyze biological networks. *BioData mining*, 4(1):10, 2011.

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

Adrian Raftery. A model for high-order markov chains. *Journal of the Royal Statistical Society: Series B (Methodological)*, 47(3):528–539, 1985.

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

Yossi Rubner, Carlo Tomasi, and Leonidas 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 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. *Generative graph models subject to global similarity*, 2020. URL https://github.com/alanadakotashine/generative_graph_models_global/.

Alana Shine and David Kempe. Generative graph models based on laplacian spectra? In *The World Wide Web Conference*, WWW '19, page 1691–1701. Association for Computing Machinery, 2019. ISBN 9781450366748.

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 Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 81–90, 2004.

Daniel 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 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 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 Wilson. Spectral generative models for graphs. In *Proceedings of the International Conference on Image Analysis and Processing*, pages 35–42, 2007.

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

Richard 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.