Machine Learning and Immersive Approaches to Graph Visualization

By

OH-HYUN KWON

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Kwan-Liu Ma, Chair

Raissa D'Souza

Martin Hilbert

Committee in Charge

2021

To my family

CONTENTS

ABSTRACT

Machine Learning and Immersive Approaches to Graph Visualization

Graphs are widely used to represent complex systems, such as social networks, power grids, and biological networks. Visualizing a graph in a node-link diagram or an adjacency matrix view allows people to intuitively understand and communicate the structural information in the data, which is especially helpful for non-experts in network analysis. However, despite decades of research, deriving effective visualizations of a graph is still a challenging task.

Computing node-link diagrams of a large graph needs a significant amount of time. Moreover, one needs to allocate significant computing resources without any hint of what the result will look like. This dissertation presents a technique that can instantly show a preview of a node-link diagram. I have developed novel graph kernels to quickly measure the structural similarities between graphs. In addition, this approach can estimate quality metrics (e.g., the number of edge crossings). The previews and estimations using my graph kernels can be obtained several orders of magnitude faster than computing the actual node-link diagrams and their quality metrics.

Different layouts (i.e., the node positions of a node-link diagram and the node ordering of an adjacency matrix view) can highlight distinct structural characteristics of the same graph. However, due to the vast design space, substantial efforts and expertise are required to create the desired layout of a graph that best suits the goal of the visualization. I present two machine learning techniques that assist users in producing effective visualizations of a graph, one for node-link diagrams and the other for adjacency matrix views. I have designed encoder-decoder architectures to learn a generative model that constructs a two-dimensional latent space of diverse layouts of a graph. By mapping a grid of generated samples, a two-dimensional latent space can be used as a what-you-see-is-what-you-get interface. This allows users to intuitively navigate and generate various visualizations by simply pointing to a location in the latent space. Thus, users can effortlessly design effective visualizations for network

analysis and communication without exhaustedly tweaking the parameters of layout algorithms.

The display and input devices affect users' productivity in analyzing data. The recent advances in immersive technologies (e.g., head-mounted displays) provide new opportunities to present and interact with data. However, most of the existing information visualization techniques, including techniques for visualizing graphs, have typically been designed for traditional desktop environments. Fundamental knowledge of designing effective graph visualization in immersive environments is still lacking. In this dissertation, I investigate layout, rendering, and interaction methods for visualizing networks in an immersive environment. The results of a user study suggested that participants completed network analysis tasks more quickly and accurately using my design than using traditional two-dimensional graph visualization in an immersive environment, especially for more difficult tasks and larger graphs.

Overall, this dissertation research addresses long-standing problems in graph visualization with fundamentally new designs and thus promises novel technologies that significantly surpass current capabilities.

ACKNOWLEDGMENTS

I want to thank Professor Kwan-Liu Ma, who provided endless support and advice during my Ph.D. journey. I would also like to thank Professor Raissa D'Souza and Professor Martin Hilbert for reviewing this dissertation and providing insightful feedback. Furthermore, I am grateful to my undergraduate advisor, Professor Kyungwon Lee, who believed in my potential and encouraged me to pursue a Ph.D.

Many thanks to my fellow VIDI members, especially Senthil Chandrasegaran, Jia-Kai Chou, Chris Muelder, Sandra Bae, Tarik Crnovrsanin, Takanori Fujiwara, and Tyson Neuroth, for their help to improve my research and for all the fun times we spent together.

Most of all, I would like to thank my family. This dissertation would not have been possible without their support and sacrifice throughout these years.

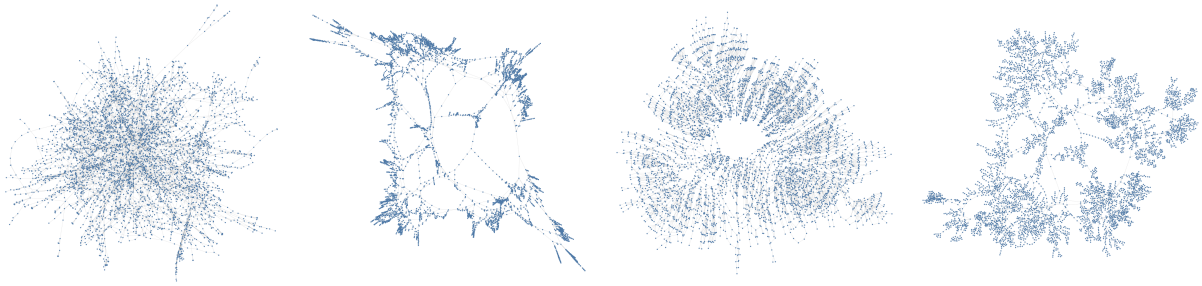
Chapter 1
Introduction

A graph, or a network, consists of a set of nodes and a set of edges where a node represents an entity, and an edge represents a relationship between two entities. Graphs are widely used to represent complex systems, such as interactions between proteins, data communications between computers, and relationships between people.

Drawing a graph as a node-link diagram, where the nodes are drawn as points, and the edges are drawn as lines, is the most popular and intuitive way to present ideas or findings on the graph. Examples include drawing of a state machine to describe and analyze a model, illustration of a molecular graph to show how the atoms are connected, and a class diagram to design a software system. The human brain has an exceptional ability to analyze visual information [169]. Hence, visualizing a graph can help better understand the relational and structural information in the data, such as patterns and outliers, that would not be as apparent if presented in a numeric form.

As drawing a node-link diagram by hand would be laborious, a multitude of methods have been devised to automatically lay out a node-link diagram since the 1960s. The layout results (i.e., the node positions) of the same graph can greatly vary depending on which method is used because each method applies a different set of rules to lay out a graph. For example, Figure 1.1 shows several layout results of the same graph.

It is important to find a "good" layout that can effectively depict the structure of a graph because the viewer's understanding of the graph can be significantly influenced



Fruchterman-Reingold [83] FM^3 [98] Kamada-Kawai [127] sfdp [116]

Figure 1.1. For different node-link diagrams of the same graph (a power grid of the western states of the United States [227], $|V| = 4,941$ $|E| = 6,594$)

by the layout of the graph [87, 118, 157, 158]. For example, due to the Gestalt principle of proximity [228], nodes that are close to each other in the layout appear to form clusters. This visual clustering can be an effective visual encoding when the nodes in a cluster have similar attributes or relationships. However, it is possible to mislead the viewer if the nodes are closely placed due to the side effects of the layout method, not because they have similar features.

Although finding the best layout of a graph is still an open problem, decades of research in this area have led to several heuristics, often called *aesthetic criteria*, for improving and evaluating the quality of a graph layout. For instance, *reducing edge crossings* has been shown as one of the most effective criteria to improve the quality of a layout [118, 132, 184, 185]. Based on aesthetic criteria, several metrics have been defined to evaluate layouts quantitatively.

However, even with aesthetic criteria and metrics, human intervention is still needed in the process of selecting a layout. While each heuristic attempts to enhance certain aspects of a layout, it does not guarantee the overall quality improvement of the layout [70, 87]. Depending on the given circumstances (e.g., given graph, task, and environment), certain criteria can lead to incomprehensible layouts [30]. To make matters worse, there is no consensus on which criteria are the most helpful in a given circumstance [70]. In fact, studies have shown conflicting and inconclusive results [87, 132]. Also, it is often not feasible to satisfy several aesthetic criteria in one layout

because conforming to one may result in violating others [66,67]. Furthermore, selecting a good layout is highly subjective, as each person might have a different opinion on what is a "good" layout, even in the same setting.

As selecting a layout depends on various factors, the user typically relies on time-consuming trial-and-error approach to find a suitable layout. This often requires computing multiple layouts and their corresponding aesthetic metrics. Since graphs are commonly used in data analysis tasks and are expected to grow in size at even higher rates, alternative solutions are needed. This dissertation addresses several challenges of visualizing graphs with fundamentally new approaches.

Enabling Quick Preview of Large Graph Visualization using a Graph Kernel

When a graph is large, it is not practical to rely on trial-and-error for selecting a suitable layout. The amount of time needed to compute various layouts and aesthetic metrics is tremendous. For example, laying out a graph with millions of nodes can take hours or days. Furthermore, analysts often have to invest in expensive computing resources without any hint of what the layout result will look like.

Chapter 2 presents a technique that can instantly show a *preview* of a graph visualization. A *preview* is obtained based on the following assumption: given the same layout method and the same parameter setting, if two graphs have similar topological structures, then they are expected to have similar layout results. Under this assumption, users can expect what a graph would look like using the chosen layout method based on the existing layouts of graphs that are structurally similar to the input graph. To achieve this, I developed novel graph kernels to measure the structural similarities between graphs. The result of a user study demonstrates that the structural similarity computed with my graph kernel matches the perceptual similarity assessed by human participants. In addition, this approach can estimate layout quality metrics (e.g., number of edge crossings). The previews and estimations using my graph kernels can be derived several orders of magnitude faster than computing the actual layouts and their quality metrics.

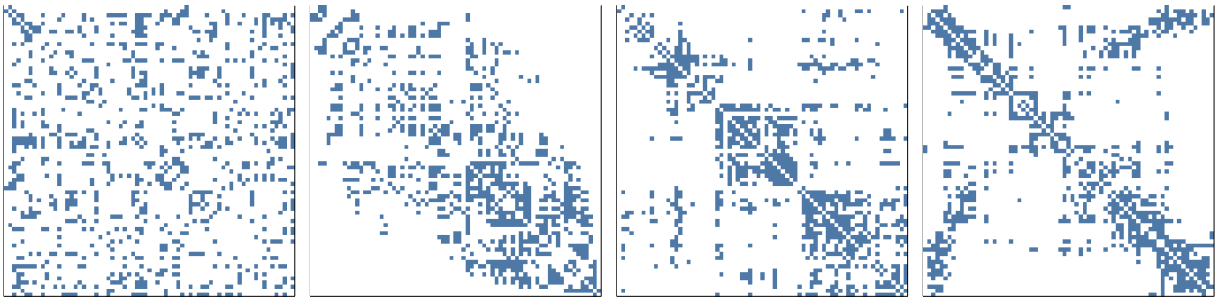
A version [145] of the research in Chapter 2 is published in IEEE Transactions on Visualization and Computer Graphics and was presented at IEEE Information Visualization Conference 2017.

Augmenting the Design Process of Node-Link Diagrams using a Deep Generative Model

Different layouts can highlight different structural characteristics of the same graph. It is thus necessary to find a "good" layout that shows the features of a graph deemed important by the user. In practice, users often visualize a graph in multiple layouts by using different methods and varying parameter settings until they find the layout that satisfies the purpose of the visualization. However, this trial-and-error process often results in a haphazard and tedious examination of a large number of layouts. Moreover, without expertise in layout algorithms, users often blindly tweak parameters, explore only a fraction of possible layouts, and thus overlook critical insights in the data.

Chapter 3 presents a machine learning technique that builds an intuitive interface for users to effortlessly produce the layout they want. I have designed an encoder-decoder architecture to learn a generative model that constructs a latent space of diverse layouts of a graph. In particular, I trained the model to construct a two-dimensional latent space. By mapping a grid of generated samples, a two-dimensional latent space can be used as a what-you-see-is-what-you-get (WYSIWYG) interface. This allows users to intuitively navigate and generate various layouts by simply pointing to a location in the latent space, without blindly tweaking parameters of layout algorithms. Thus, users can easily design effective graph visualizations without a haphazard trial-and-error process or any expertise in graph layout algorithms. Furthermore, the evaluation results show that the model is capable of learning and generalizing abstract concepts of graph layouts, not just memorizing the training examples.

A version [146] of the research in Chapter 3 is published in IEEE Transactions on Visualization and Computer Graphics and was presented at IEEE Information Visualization Conference 2019.



VAT [28] Spectral [68] HC-Average [102] QAP-BAR [73, 102]

Figure 1.2. Four different matrix reorderings of the same graph (the macaque cortical connectivity [240], $|V| = 71$ $|E| = 438$).

Augmenting the Design Process of Adjacency Matrix Views using a Deep Generative Model

In addition to a node-link diagram, another popular approach to graph visualization is an adjacency matrix view, where the nodes are laid out as the rows and the columns of the matrix, and an edge between two nodes is represented by coloring the corresponding cell in the matrix. Adjacency matrix views are often preferred for visualizing dense graphs as they do not suffer from occlusion, which is common in node-link diagrams (e.g., node overlaps and edge crossings).

When visualizing a graph as node-link diagrams, different node positions can show different structural information of the same graph. Likewise, in adjacency matrix views, different node orderings often highlight different structural patterns of the same graph (e.g., Figure 1.2). Finding a "good" node ordering is thus a critical step in designing effective matrix visualization of a graph. However, users often try different matrix reorderings using different methods until they find one that best fits the purpose of the visualization, similar to how users typically visualize a graph in node-link diagrams.

Chapter 4 presents a technique that helps users to intuitively find the desired matrix view of a graph. The fundamental approach is similar to the one introduced in Chapter 3: (1) train a generative model that constructs a latent space of diverse matrix reorderings of a graph and (2) build a WYSIWYG interface using the learned latent space for users to effortlessly explore various matrix reorderings of the given graph.

However, designing a neural network architecture for reordering an adjacency matrix faces unique challenges. For example, reordering a matrix is not a differentiable operation. In addition, there can be different node orderings that produce the equivalent matrix reordering due to the automorphisms. Chapter 4 introduces a design of a neural network architecture addressing these challenges and demonstrates its capability to learn a generative model that produces diverse matrix reorderings of the given graph.

Designing Graph Visualization for Immersive Environments

Graph visualization has traditionally limited itself to two-dimensional representations, primarily due to the prevalence of two-dimensional displays and report formats. However, there has been a recent surge in popularity of consumer grade immersive displays, such as head-mounted displays. While techniques that utilize such immersive environments have been explored extensively for spatial and scientific visualizations, contrastingly very little has been explored for graph visualization.

Chapter 5 investigates layout, rendering, and interaction methods for visualizing graphs in an immersive environment. I conducted a user study to evaluate my techniques compared to traditional two-dimensional graph visualization in an immersive environment. The results show that participants answered significantly faster with a fewer number of interactions using my techniques, especially for more difficult tasks. While the overall correctness rates are not significantly different, participants gave significantly more correct answers using my techniques for larger graphs.

A version [148] of the research in Chapter 5 is published in IEEE Transactions on Visualization and Computer Graphics and was presented at IEEE Information Visualization Conference 2016. Also, an initial work [147] of the research was presented at IEEE Pacific Visualization Symposium 2015.

Chapter 2

Enabling Quick Preview of Large Graph Visualization using a Graph Kernel

When the graph is large, computing several graph layouts and selecting one through visual inspection and/or aesthetic metrics is, unfortunately, not a practical solution. The amount of time it would take to compute these various layouts and aesthetic metrics is tremendous. For a graph with millions of nodes, a single layout can take several hours or even days to calculate. In addition, we often must consider multiple aesthetic metrics to evaluate a single graph layout, since there is no consensus on which criteria are the most effective or preferable [70].

One possible solution to help a time-consuming trial-and-error for selecting a layout is to quickly estimate aesthetic metrics and show what a graph would look like through predictive methods. In the field of machine learning, several methods have been used to predict the properties of graphs, such as the classes of graphs. One prominent approach to predicting such properties is to use a graph kernel. Graph kernel methods enable us to apply various kernelized machine learning techniques, such as the Support Vector Machine (SVM) [55], on graphs.

This chapter presents a machine learning approach that can instantly show what a graph would look like in different layouts and estimate their corresponding aesthetic

metrics. The fundamental assumption of our approach is the following: given the same layout method, if the graphs have similar topological structures, then they will have similar resulting layouts. Under this assumption, we introduce new graph kernels to measure the topological similarities between graphs. Then, we apply machine learning techniques to show what a new input graph would look like in different layouts and estimate their corresponding aesthetic metrics. To the best of our knowledge, this is the first time graph kernels have been utilized in the field of graph visualization.

The primary contributions of this chapter include:

- A fast and accurate method to show what a graph would look like in different layouts.
- A fast and accurate method to estimate graph layout aesthetic metrics.
- A framework for designing graph kernels based on graphlets.
- A demonstration of the effectiveness of graph kernels as an approach to large graph visualization.

We evaluate our methods in two ways. First, we compare 13 graph kernels, which include two state-of-the-art ones, based on accuracy and computation time for estimating aesthetic metrics. The results show that our estimations of aesthetic metrics are highly accurate and fast. Our graph kernels outperform existing ones in both time and accuracy. Second, we conduct a user study to demonstrate that the topological similarity computed with our graph kernel matches the perceptual similarity assessed by human.

2.1 Background

This section presents the notations and definitions used in this chapter and introduce graph kernels.

2.1.1 Notations and Definitions

Let $G = (V, E)$ be a graph, where $V = \{v_1, \ldots, v_n\}$ is a set of n nodes (or vertices), and $E = \{e_1, \ldots, e_m \mid e = (v_i, v_j),\ v_i, v_j \in V\}$ is a set of m edges (or links). An edge $e = (v_i, v_j)$ is said to be *incident* to node v_i and node v_j. An edge that connects a node to itself $e = (v_i, v_i)$ is called a *self loop*. Two or more edges that are incident to the same two

nodes are called *multiple edges*. A graph is considered *simple* if it contains no self loops or multiple edges. An *undirected* graph is one where $(v_i, v_j) \in E \Leftrightarrow (v_j, v_i) \in E$. A graph is called *unlabeled* if there is no distinction between nodes other than their interconnectivity. In this chapter, we consider simple, connected, undirected, and unlabeled graphs.

Given a graph G, a graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A subgraph $G' = (V', E')$ is called an *induced (or vertex-induced)* subgraph of G if $E' = \{(v_i, v_j) \mid (v_i, v_j) \in E \text{ and } v_i, v_j \in V'\}$, that is, all edges in E, between two nodes $v_i, v_j \in V'$, are also present in E'. Two graphs $G = (V, E)$ and $G' = (V', E')$ are *isomorphic* if there exists a bijection $f : V \to V'$, called *isomorphism*, such that $(v_i, v_j) \in E \Leftrightarrow (f(v_i), f(v_j)) \in E'$ for all $v_i, v_j \in V$.

Suppose we have empirical data $(x_1, y_1), \ldots, (x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$, where the domain $\mathcal{X}$ is a nonempty set of *inputs* x_i and $\mathcal{Y}$ is a set of corresponding *targets* y_i. A kernel method predicts the target y of a new input x based on existing "similar" inputs and their outputs (x_i, y_i). A function $k : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ that measures the similarity between two inputs is called a kernel function, or simply a kernel. A kernel function k is often defined as an inner product of two vectors in a *feature space* $\mathcal{H}$:

$$k(x, x') = \langle \phi(x), \phi(x') \rangle = \langle \mathbf{x}, \mathbf{x}' \rangle$$

where $\phi : \mathcal{X} \mapsto \mathcal{H}$ is called a *feature map* which maps an input x to a *feature vector* $\mathbf{x}$ in $\mathcal{H}$.

2.1.2 Measuring Topological Similarities between Graphs

Based on our assumption, we need to measure the topological similarities between graphs. Depending on the discipline, this problem is called *graph matching*, *graph comparison*, or *network alignment*. In the last five decades, numerous approaches have been proposed to this problem [64, 76]. Each approach measures the similarity between graphs based on different aspects, such as isomorphism relation [126, 203, 242], graph edit distance [40], or graph measures [17, 56]. However, many of these traditional approaches are either computationally expensive, not expressive enough to capture the topological features, or difficult to adapt to different problems [173].

Graph kernels have been recently introduced for measuring pairwise similarities

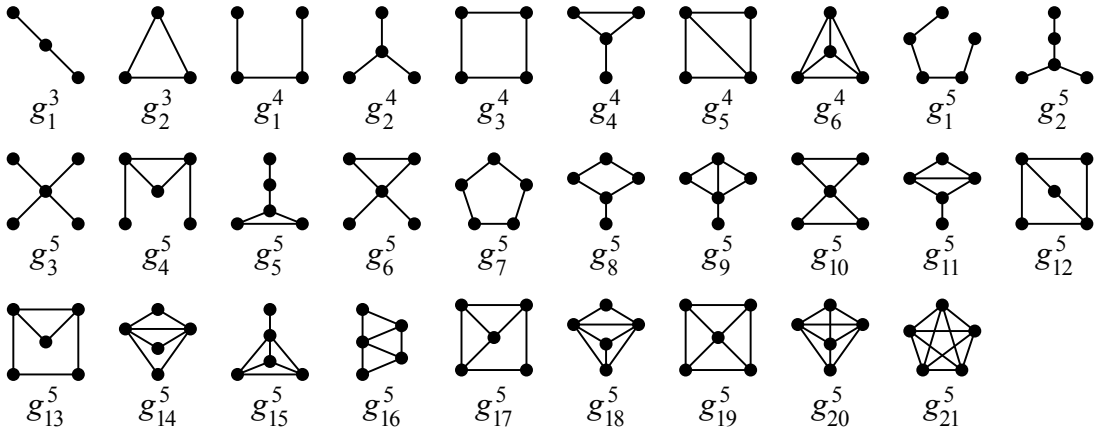


Figure 2.1. All connected graphlets of 3, 4, or 5 nodes.

between graphs in the field of machine learning. They allow us to apply many different kernelized machine learning techniques, e.g. SVM [55], on graph problems, including graph classification problems found in bioinformatics and chemoinformatics.

A graph kernel can be considered to be an instance of an R-convolution kernel [108]. It measures the similarity between two graphs based on the recursively decomposed substructures of said graphs. The measure of similarity varies with each graph kernel based on different types of substructures in graphs. These substructures include walks [86,130,217], shortest paths [31], subtrees [189,196,197], cycles [115], and graphlets [198]. Selecting a graph kernel is challenging as many kernels are available. To exacerbate the problem, there is no theoretical justification as to why one graph kernel works better than another for a given problem [198].

Many graph kernels often have similar limitations as previously mentioned approaches. They do not scale well to large graphs (time complexity of $O(|V|^3)$ or higher) or do not work well for unlabeled graphs. To overcome this problem, a graph kernel based on sampling a fixed number of graphlets has been introduced to be accurate and efficient on large graphs [173,198].

Graphlets are small, induced, and non-isomorphic subgraph patterns in a graph [182] (Figure 2.1). Graphlet frequencies (Figure 2.2) have been used to characterize biological networks [181,182], identify disease genes [163], and analyze social network structures [211]. Depending on the definition, the relative frequencies are called *graphlet*

frequency distribution, *graphlet degree distribution*, or *graphlet concentrations*. While these works often have different definitions, the fundamental idea is to count the individual graphlets and compare their relative frequencies of occurrence between graphs. A graph kernel based on graphlet frequencies, called a graphlet kernel, was first proposed by Shervashidze et al. [198]. The main idea is to use a graphlet frequency vector as the feature vector of a graph, then compute the similarity between graphs by defining the inner product of the feature vectors.

2.2 Approach

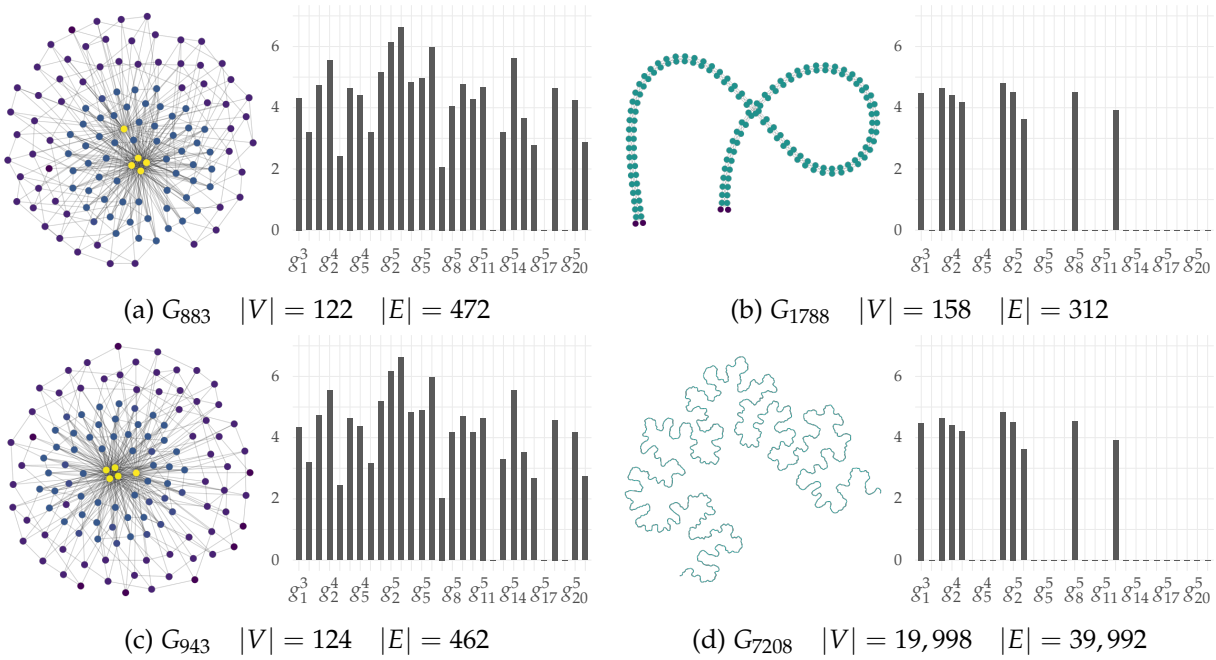
Our approach is a supervised learning of the relationship between topological features of existing graphs and their various layout results. This also includes the layouts' aesthetic metrics. Like many supervised learning methods, our approach requires empirical data (training data) of existing graphs, their layout results, and corresponding aesthetic metrics. This generally takes a considerable amount of time, but can be considered a preprocessing step as it is only ever done once. The benefit of machine learning is that as we add more graphs and their layout results, the performance generally improves. In this section, we introduce:

1. A framework for designing better graphlet kernels
2. A process of using graph kernels to determine what a graph would look like in different layouts
3. A method to estimate the aesthetic metrics without calculating actual layouts

2.2.1 Graphlet Kernel Design Framework

One of the key challenges of our approach is choosing a graph kernel. While many graph kernels are available, we focus on sampling based graphlet kernels because they are computationally efficient and are designed for unlabeled graphs. To improve the performance of a graphlet kernel, we introduce a framework for designing graphlet kernels. Our framework consists of three steps:

1. Sampling graphlet frequencies
2. Scaling graphlet frequency vectors



(a) G_{883} $|V| = 122$ $|E| = 472$ (b) G_{1788} $|V| = 158$ $|E| = 312$

(c) G_{943} $|V| = 124$ $|E| = 462$ (d) G_{7208} $|V| = 19,998$ $|E| = 39,992$

Figure 2.2. Examples of graphlet frequencies. The x-axis represents connected graphlets of size $k \in \{3, 4, 5\}$ and the y-axis represents the weighted frequency of each graphlet. Four graphs are drawn with sfdp layouts [116]. If two graphs have similar graphlet frequencies, i.e., high topological similarity, they tend to have similar layout results (a and c). If not, the layout results look different (a and b). However, in rare instances, two graphs can have similar graphlet frequencies (b and d), but vary in graph size, which might lead to different looking layouts.

3. Defining an inner product between two graphlet frequency vectors

For each step, we discuss several possible design choices. Our framework defines several new types of graphlet kernels.

Existing studies related to graphlets and graphlet kernels are scattered throughout the literature, including machine learning and network science. Each of the studies focuses on certain aspects of a graphlet kernel. Our framework unifies the related works for graphlet kernels.

2.2.1.1 Sampling Graphlet Frequencies

One of the challenges for constructing a graphlet kernel is computing the graphlet frequencies. Exhaustive enumeration of graphlets with k nodes in a graph G is $O(|V|^k)$, which is prohibitively expensive, even for graphs with a few hundred or more nodes. Thus, sampling approaches have been introduced to obtain the graphlet frequencies in

a short amount of time with acceptable error.

Random node sampling (RN): Most existing works on graphlet kernels have sampled graphlets using a random node sampling method [32]. To sample a graphlet of size k, this method randomly chooses k nodes in V and induces the graphlet based on their adjacency in G. This step is repeated until the number of sampled graphlets is sufficient. Since this sampling method randomly chooses k nodes without considering their interconnectivity in G, it has several limitations. As many real world networks are sparse [17], $|E| \ll O(|V|^2)$, most randomly sampled graphlets are disconnected. Consequently, the frequency of disconnected graphlets would be much higher than connected ones. If disconnected graphlets lack discriminating traits between graphs, and they outnumber the informative graphlets, comparing graphs becomes increasingly difficult. While we could only sample connected graphlets by excluding disconnected ones, this requires a tremendous number of sampling iterations to sample a sufficient number of connected graphlets. Since there is no lower bound on the number of iterations for sampling certain amounts of connected graphlets, using RN to sample only connected graphlets would lead to undesirable computation times.

Random walk sampling (RW): There are other methods to sample graphlets based on random walks, such as Metropolis-Hasting random walk [187], subgraph random walk [220], and expanded Markov chain [48]. However, they have not been used for designing graphlet kernels in the machine learning community. Unlike RN sampling, they sample graphlets by traversing the structure of a given graph. That is, they search for the next sample nodes within the neighbors of the currently sampled nodes. Thus, they are able to sample connected graphlets very well.

2.2.1.2 Scaling Graphlet Frequency Vector

A graphlet frequency vector $\mathbf{x}$ is defined such that each component x_i corresponds to the relative frequency of a graphlet g_i. In essence, the graphlet frequency vector of a graph is the feature vector of the graph.

Linear scale (LIN): Many existing graphlet kernels use linear scaling, often called *graphlet concentration*, which is the percentage of each graphlet in the graph. As several

works use weighted counts w_i of each graphlet g_i, this scaling can be defined as:

$$x_i = \frac{w_i}{\sum w_i} \tag{2.1}$$

Logarithmic scale (LOG): Similar to the node degree distribution, the distribution of graphlet frequency often exhibits a power-law distribution. This again can cause a significant problem if graphlets that lack discriminating traits between graphs outnumber the informative graphlets. Thus, several studies [182, 187] used a logarithmic scale of the graphlet frequency vector to solve this problem. While the exact definition of these methods differ, we generalize it using the following definition:

$$x_i = \log\left(\frac{w_i + w_b}{\sum(w_i + w_b)}\right) \tag{2.2}$$

where $w_b > 0$ is a base weight to prevent $\log 0$.

2.2.1.3 Defining Inner Product

Several kernel functions can be used to define the inner product in a feature space $\mathcal{H}$.

Cosine similarity (COS): Most existing graphlet kernels use the dot product of two graphlet frequency vectors in Euclidean space, then normalize the kernel matrix. This is equivalent to the cosine similarity of two vectors, which is the L_2-normalized dot product of two vectors:

$$\left\langle \mathbf{x}, \mathbf{x}' \right\rangle = \frac{\mathbf{x} \cdot \mathbf{x}'^{\mathsf{T}}}{\|\mathbf{x}\| \, \|\mathbf{x}'\|} \tag{2.3}$$

Gaussian radial basis function kernel (RBF): This kernel is popularly used in various kernelized machine learning techniques:

$$\left\langle \mathbf{x}, \mathbf{x}' \right\rangle = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right) \tag{2.4}$$

where σ is a free parameter.

Laplacian kernel (LAPLACIAN): Laplacian kernel is a variant of RBF kernel:

$$\left\langle \mathbf{x}, \mathbf{x}' \right\rangle = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_1}{\sigma}\right) \tag{2.5}$$

where $\|x - x'\|_1$ is the L_1 distance, or Manhattan distance, of the two vectors.

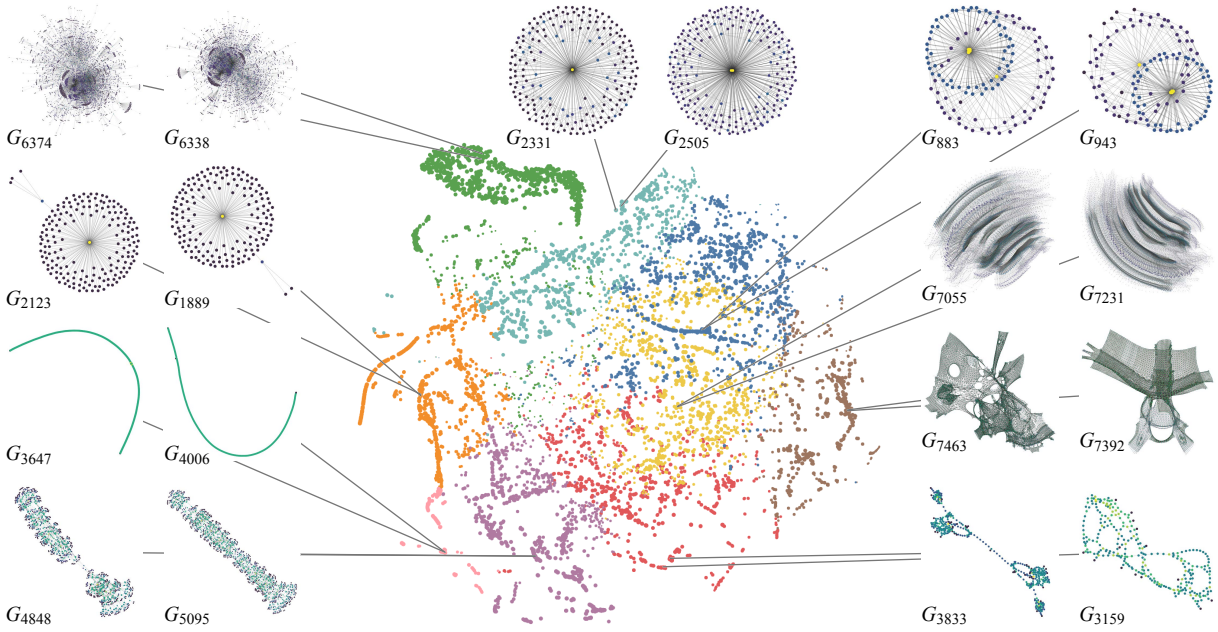


Figure 2.3. A projection of topological similarities between 8,263 graphs measured by our RW-LOG-LAPLACIAN kernel. Based on the topological similarity, our approach shows what a graph would look like in different layouts and estimates their corresponding aesthetic metrics. The graphs are clustered based on their topological similarities for the purpose of the user study. The two graphs in each pair are the most topologically similar, but not isomorphic, to each other. The projection is computed with t-SNE [214], and the highlighted graphs are visualized with FM^3 layout [98]. An interactive plot is available in the supplementary materials [9].

2.2.2 What Would a Graph Look Like in This Layout? (WGL)

Graph kernels provide us with the topological similarities between graphs. Using these pairwise similarities, we design a nearest neighbors based method, similar to k-nearest neighbors, to show what a graph would look like in different layouts. Given a new input graph G_{input}, we find the k most topologically similar graphs, and show their existing layout results to the users. Thus, if our assumption is true, users can expect the layout results of new input graphs by looking at the layout results of topologically similar graphs.

Graph kernels can fail to capture the similarity between graphs depending on the structural characteristics of the given graphs. Some cases with obvious causes can be addressed by adding a set of constraints in the process of finding k topologically most

similar graphs. For example, while graph kernels are able to find topologically similar graphs, many of them do not explicitly take the size of graphs into account. In rare cases, it is possible to find topologically similar graphs that vary in size. For instance, 2.2b and 2.2d have similar graphlet frequencies, yet have different layout results. To prevent this, we add some constraints when we find similar graphs, such as only counting a graph that has more than half and less than double the amount of nodes as the input graph. Although many constraints could be added, in the evaluation (Section 2.4), we only use this size-based constraint to evaluate the raw power of the graph kernels.

As a result, for the new input graph G_{input}, we find k most similar graphs as follows:

1. Compute the similarity between existing graphs and G_{input}
2. Remove the graphs that do not satisfy a set of constraints
3. Select the k most similar graphs

After we have obtained the k most similar graphs to the input graph G_{input}, we show their existing layout results to the user.

2.2.3 Estimating Aesthetic Metrics (EAM)

As the aesthetic metrics are continuous values, estimating the aesthetic metrics is a regression problem. There are several kernelized regression models, such as Support Vector Regression (SVR) [202], that can be used for estimating the aesthetic metrics based on the similarities between graphs obtained by graph kernels. Computing actual aesthetic metrics of a layout requires a calculation of the layout first. However, our approach is able to estimate the metrics without calculating actual layouts.

Training: To estimate the aesthetic metrics, we first need to train a regression model:

1. Prepare the training data (layouts and their aesthetic metrics of existing graphs)
2. Compute a kernel matrix using a graph kernel (pairwise similarities between all graphs in the training data)
3. Train regression model

Estimation: To estimate an aesthetic metric of a new input graph:

1. Compute similarity between the input graph and other graphs in the training data
2. Estimate value using the trained regression model

2.3 Evaluation 1: Estimating Layout Aesthetic Metrics

There are several questions we wanted to answer within this evaluation:

- Is our method able to accurately estimate the layout's aesthetic metrics without computing the layout?
- Is our method able to quickly obtain the estimations?
- Does our graph kernels, derived from our framework, outperform state-of-the-art graph kernels in terms of computation time and estimation accuracy?

We describe the experimental design, apparatus, implementation, and metrics used in the study. We also answer each of these questions in the results section.

2.3.1 Experimental Design

We perform 10-fold cross validations to compare 13 different graph kernels in terms of their accuracy and computation times for estimating four aesthetic metrics on eight layout methods.

2.3.1.1 Datasets

We started by collecting around 3,700 graphs from [61], which includes, but not limited to, social networks, web document networks, and geometric meshes. Without loss of generality, graphs with multiple connected components were broken down into separate graphs (one connected component to one graph). After that, we removed any graph with less than 100 nodes, as there would be little benefit from a machine learning approach since most layout methods are fast enough for small graphs. This left us with a total of 8,263 graphs for our study. The graphs range from 100 nodes, and 100 edges up to 113 million nodes and 1.8 billion edges. More details about the graphs, such as characteristic measures and layout results, can be found in the supplementary materials [9].

Not all graphs were used for each layout method, as some layout algorithms failed to compute the results within a reasonable amount of time (10 days) or ran out of memory. Exact numbers of graphs used for each layout are reported in the supplementary materials [9].

2.3.1.2 Kernels

We compare a total of 13 graphlet kernels. Using our framework, 12 graphlet kernels are derived from a combination of 2 graphlet sampling methods (RN and RW) $\times$ 2 types of graphlet frequency vector scaling (LIN and LOG) $\times$ 3 inner products (COS, RBF, and LAPLACIAN). We denote a kernel derived from our framework by a combination of the above abbreviations. For example, RW-LOG-LAPLACIAN denotes a graphlet kernel which samples graphlets based on a random walk, uses a log scaled graphlet frequency vector, and computes the similarity using the Laplacian kernel function. We also compare with state-of-the-art graphlet kernels. The original graphlet kernel [198] can be constructed using our framework and is included in the 12 kernels (RN-LIN-COS). Lastly, the 13^{th} graph kernel is a Deep Graphlet Kernel (DGK) from Yanardag and Vishwanathan [236].

We used a sampling method from Chen et al. [48] as the RW sampling. For all kernels, we sampled 10,000 graphlets of 3, 4, and 5 nodes for each graph. The graphlets of 3, 4, and 5 nodes are widely used due to computational costs. Also, the graphlets of 6 or more nodes are rare [187]. RW sampling considers only connected graphlets, as in [48], while RN sampling counts both connected and disconnected graphlets, as in [198, 236]. All kernel matrices are normalized such that the similarity between a graph and itself has a value of 1.

2.3.1.3 Layouts

The process of laying out a graph has been actively studied for over five decades. Several studies [66,67,87,206,218] provide a comprehensive review of these layout methods. While there are methods designed for specific purposes, such as orthogonal layout methods [132], in this chapter, we focus on two-dimensional layout methods that draw all edges in straight lines. Due to the volume of layout methods proposed, evaluating all the methods is impractical. For this evaluation, we used eight representative layout methods of five families based on groupings found in [87, 218]. Our selection process prioritized methods that have been popularly used, have a publicly available implementation, and have a proven track record over other state-of-the-art methods.

Force-directed methods: Force-directed methods are based on a physical model of attraction and repulsion. They are among the first layout methods to be developed and are some of the most commonly used layout methods today. In general, force-directed layout methods fall within two groups: spring-electrical [71,81,83] and energy-based approaches [60,127]. We selected one from each group: Fruchterman-Reingold (FR) [83] from spring-electrical approaches and Kamada-Kawai (KK) [127] from energy-based approaches.

Dimensionality reduction based method: Dimensionality reduction techniques, including multidimensional scaling (MDS) or principal component analysis (PCA), can be used to lay out a graph using the graph-theoretic distance between node pairs. PivotMDS [35] and High-Dimensional Embedder (HDE) [107] work by assigning several nodes as the pivots, then constructing a matrix representation with graph-theoretic distances of all nodes from the pivots. Afterward, dimensionality reduction techniques are applied to the matrix. We selected HDE [107] in this family.

Spectral method: Spectral layout methods use the eigenvectors of a matrix, such as the distance matrix [51] or the Laplacian matrix [140] of the graph, as coordinates of the nodes. We selected the method by Koren [140] in this family.

Multi-Level methods: Multilevel layout methods are developed to reduce computation time. These multilevel methods hierarchically decompose the input graph into coarser graphs. Then, they lay out the coarsest graph and use the node position as the initial layout for the next finer graph. This process is repeated until the original graph is laid out. Several methods can be used to lay out the coarse graph, such as a force-directed method [85,98,106,116,219], a dimensionality reduction based method [54], or a spectral method [82,141]. We selected sfdp [116] and FM^3 [98] in this family.

Clustering based methods: Clustering based methods are designed to emphasize graph clusters in a layout. When the graph size is large, users tend to ignore the number of edge crossings in favor of well-defined clusters [215]. We selected the Treemap based layout [165] and the Gosper curve based layout [166] from this family, which utilizing the hierarchical clustering of a graph to lay out the graph.

2.3.1.4 Aesthetic Metrics

Aesthetic criteria, e.g., minimizing the number of edge crossings, are used for improving the readability of a graph layout. Bennett et al. [25] reviewed various aesthetic criteria from a perceptual basis. Aesthetic metrics enable a quantitative comparison of the aesthetic quality of different layouts. While there are many aesthetic criteria and metrics available, many of them are informally defined or are defined only for specific types of layout methods. Many also do not have a normalized metric for comparing graphs of different sizes, or are too expensive to compute (e.g., symmetry metric in [183] is $O(n^7)$). In this evaluation, we chose four aesthetic metrics because they have a normalized form, are not defined only for specific types of layouts, and can be computed in a reasonable amount of time.

Crosslessness (m_c) [183]: *Minimizing the number of edge crossings* has been found as one of the most important aesthetic criteria in many studies [118, 132, 184, 185]. The crosslessness m_c is defined as

$$m_c = \begin{cases} 1 - \dfrac{c}{c_{\max}}, & \text{if } c_{\max} > 0 \\ 1, & \text{otherwise} \end{cases} \tag{2.6}$$

where c is the number of edge crossings and $c_{\max}$ is the approximated upper bound of the number of edge crossings, which is defined as

$$c_{\max} = \frac{|E|\,(|E|-1)}{2} - \frac{1}{2}\sum_{v \in V}\left(\deg(v)\,(\deg(v)-1)\right) \tag{2.7}$$

Minimum angle metric (m_a) [183]: This metric quantifies the criteria of *maximizing the minimum angle between incident edges on a node*. It is defined as the average absolute deviation of minimum angles between the incident edges on a node and the ideal minimum angle $\theta(v)$ of the node:

$$m_a = 1 - \frac{1}{|V|}\sum_{v \in V}\left|\frac{\theta(v) - \theta_{\min}(v)}{\theta(v)}\right|, \quad \theta(v) = \frac{360^\circ}{\deg(v)} \tag{2.8}$$

where $\theta_{\min}(v_i)$ is the minimum angle between the incident edges on the node.

Edge length variation (m_l) [99]: *Uniform edge lengths* have been found to be effective aesthetic criteria for measuring the quality of a layout in several studies [132]. The

coefficient of variance of the edge length (l_{cv}) has been used to quantify this criteria [99]. Since the upper bound of the coefficient of variation of n values is $\sqrt{n-1}$ [131], we divide l_{cv} by $\sqrt{|E|-1}$ to normalize:

$$m_l = \frac{l_{\text{cv}}}{\sqrt{|E|-1}}, \;\; l_{\text{cv}} = \frac{l_\sigma}{l_\mu} = \sqrt{\frac{\sum\limits_{e \in E} (l_e - l_\mu)^2}{|E| \cdot l_\mu^2}} \tag{2.9}$$

where l_σ is the standard deviation of the edge length and l_μ is the mean of the edge length.

Shape-based metric (m_s) [72]: Shape-based metric is a more recent aesthetic metric and was proposed for evaluating the layouts of large graphs. The shape-based metric m_s is defined by the mean Jaccard similarity (MJS) between the input graph G_{input} and the shape graph G_{S}:

$$m_s = \text{MJS}(G_{\text{input}}, G_{\text{S}}), \;\; \text{MJS}(G_1, G_2) = \frac{1}{|V|} \sum_{v \in V} \frac{|N_1(v) \cap N_2(v)|}{|N_1(v) \cup N_2(v)|} \tag{2.10}$$

where $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ are two graphs with the same node set and $N_i(u)$ is the set of neighbours of v in G_i. We use the Gabriel graph [84] as the shape graph.

2.3.2 Apparatus and Implementation

We perform 10-fold cross validations of ϵ-SVR implemented by [45]. To remove random effects of the fold assignments, we repeat the whole experiment 10 times and report mean accuracy metrics (Table 2.1).

We obtained the implementation of DGK [236] from the authors. The implementation of each layout method was gathered from: sfdp [176], FM^3 [49], FR [176], KK [49], Spectral [100], and Treemap [165] and Gosper [166] are provided by the authors. Other kernels and layout methods were implemented by us. For crosslessness (m_c), a GPU based massively parallel implementation was used. For other metrics, parallel CPU based on implementations written in C++ were used. The machine we used to generate the training data and to conduct the experiment has two Intel Xeon processors (E5-4669 v4) with 22 cores (2.20 GHz) each, and two NVIDIA Titan X (Pascal) GPUs .

2.3.3 Accuracy Metrics

Root-Mean-Square Error (RMSE) measures the difference between measured values (ground truth) and estimated values by a model. Given a set of measured values $\mathcal{Y} = \{y_1, \ldots, y_n\}$ and a set of estimated values $\tilde{\mathcal{Y}} = \{\tilde{y_1}, \ldots, \tilde{y_n}\}$, the RMSE is defined as:

$$\text{RMSE}(\mathcal{Y}, \tilde{\mathcal{Y}}) = \sqrt{\frac{1}{n}\sum_i (y_i - \tilde{y}_i)^2} \tag{2.11}$$

The coefficient of determination (R^2) shows how well a model "fits" the given data. The maximum R^2 score is 1.0 and it can have an arbitrary negative value. Formally, it indicates the proportion of the variance in the dependent variable that is predictable from the independent variable, which is defined as:

$$R^2(\mathcal{Y}, \tilde{\mathcal{Y}}) = 1 - \sum_i (y_i - \tilde{y}_i)^2 \Big/ \sum_i (y_i - y_\mu)^2 \tag{2.12}$$

where y_μ is the mean of measured values y_i.

2.3.4 Results

We report the accuracy and computation time for estimating the aesthetic metrics.

2.3.4.1 Estimation Accuracy

Due to space constraints, we only reported the results of the two most accurate kernels and state-of-the-art kernels [198, 236] in Table 2.1. The results shown are mean RMSE (lower is better) and mean R^2 (higher is better) from 10 trials of 10-fold cross validations. The standard deviations of RMSE and R^2 are not shown because the values are negligible: all standard deviations of RMSE are lower than .0006 and all standard deviations of R^2 are lower than .0075. We ranked the kernels based on the mean RMSE of all estimations.

The most accurate kernel is our RW-LOG-LAPLACIAN kernel. Except for the crosslessness (m_c) of FM^3, our RW-LOG-LAPLACIAN kernel shows best estimation results in both RMSE and R^2 score for all four aesthetic metrics on all eight layout methods. The second most accurate kernel is our RW-LOG-RBF kernel and is the best for crosslessness (m_c) of FM^3.

Our RW-LOG-LAPLACIAN kernel (mean RMSE = .0557 and mean R^2 = .8169) shows an average of 2.46 times lower RMSE than existing kernels we tested. The original graphlet kernel [198] (mean RMSE = .1366 and mean R^2 = .1216), which is denoted as RN-LIN-COS, ranked 11^{th}. The DGK [236] (mean RMSE = .1388 and mean R^2 = .0540) ranked 12^{th}.

Within the kernels we derived, the kernels using RW sampling show higher accuracy (mean RMSE = .0836 and mean R^2 = .6247) than ones using RN sampling (mean RMSE = .1279 and mean R^2 = .2501). The kernels using LOG scaling show higher accuracy (mean RMSE = .0955 and mean R^2 = .5279) than ones using LIN (mean RMSE = .1160 and mean R^2 = .3469). The kernels using LAPLACIAN as the inner product show higher accuracy (mean RMSE = .0956 and mean R^2 = .5367) than ones using RBF (mean RMSE = .1047 and mean R^2 = .4393) and COS (mean RMSE = .1169 and mean R^2 = .3362).

2.3.4.2 Computation Time

Since some algorithms are implemented in parallel while others are not, we report CPU times. The estimation time is comprised of the computation steps required for estimation in Section 2.2.3. Our RW-LOG-LAPLACIAN kernel, which shows the best estimation accuracy, also shows the fastest computation time for estimation. On average, it takes .14093 seconds (SD = 1.9559) per graph to make the estimations. Figure 2.4 shows the computation time for layouts and their aesthetic metrics.

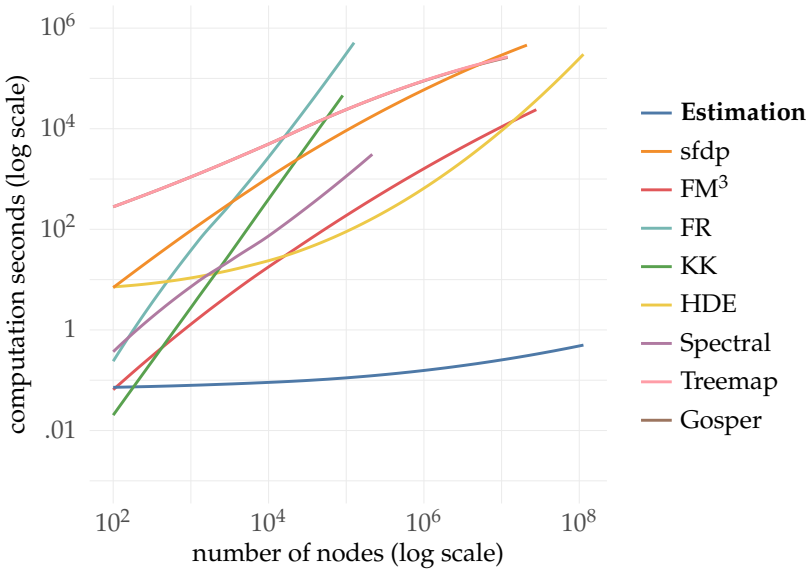
Most of the time spent on estimation was devoted to sampling graphlets. The RW sampling [48] shows fastest computation time, with an average of .14089 seconds (SD = 1.9559) per graph. The graphlet sampling for DGK take longer than RW, with an average of 3.38 seconds (SD = 7.88) per graph. The RN sampling take the longest time, with an average of 6.81 seconds (SD = 7.04).

2.3.5 Discussion

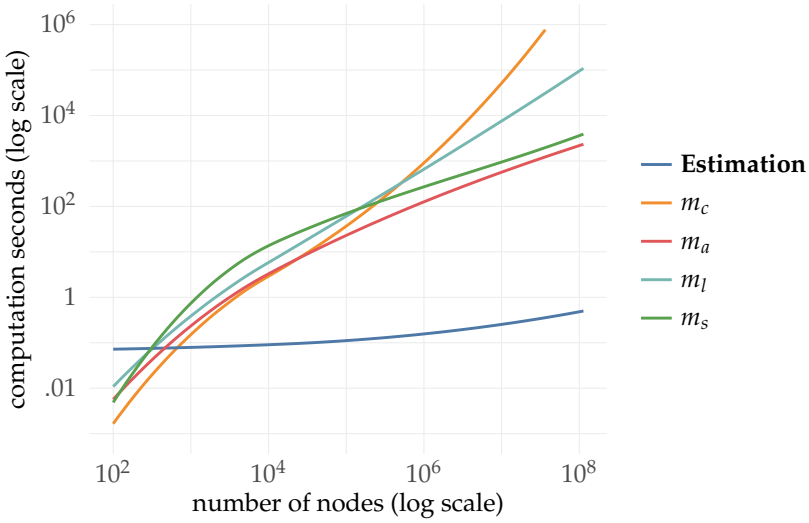
Our graph kernels perform exceptionally well in both estimation accuracy and computation time. Specifically, RW-LOG-LAPLACIAN outperforms all other kernels in all metrics except crosslessness (m_c) on the FM^3. RW-LOG-RBF is the best performing kernel on FM^3's crosslessness (m_c) and is the second best kernel. Existing graph kernels

Kernel		sfdp		FM^3		FR		KK		Spectral		HDE		Treemap		Gosper	
		RMSE	R^2	RMSE	R^2	RMSE	R^2	RMSE	R^2	RMSE	R^2	RMSE	R^2	RMSE	R^2	RMSE	R^2
Rank 1 RW-Log-Laplacian	m_c	**.0175**	**.9043**	.0468	.7319	**.0257**	**.8480**	**.0346**	**.8223**	**.1120**	**.6947**	**.0903**	**.8130**	**.0836**	**.6399**	**.0857**	**.6199**
	m_a	**.1011**	**.8965**	**.1041**	**.8919**	**.0982**	**.9004**	**.1024**	**.8876**	**.1153**	**.8793**	**.1152**	**.8666**	**.1053**	**.8552**	**.1071**	**.8580**
	m_l	**.0055**	**.9021**	**.0048**	**.8531**	**.0055**	**.9028**	**.0105**	**.4549**	**.0505**	**.6203**	**.0155**	**.5961**	**.0047**	**.8666**	**.0066**	**.8444**
	m_s	**.0514**	**.9060**	**.0474**	**.9325**	**.0417**	**.8533**	**.0485**	**.9084**	**.0534**	**.9031**	**.0486**	**.8942**	**.0112**	**.8429**	**.0323**	**.7495**
Rank 2 RW-Log-RBF	m_c	.0176	.9036	**.0446**	**.7568**	.0279	.8218	.0350	.8182	.1138	.6845	.0917	.8072	.0841	.6356	.0882	.5976
	m_a	.1070	.8840	.1102	.8788	.1023	.8920	.1061	.8793	.1193	.8706	.1202	.8546	.1101	.8416	.1125	.8434
	m_l	.0062	.8793	.0050	.8412	.0059	.8874	.0106	.4497	.0519	.5992	.0167	.5291	.0052	.8417	.0073	.8127
	m_s	.0556	.8900	.0542	.9116	.0459	.8227	.0547	.8833	.0576	.8875	.0537	.8708	.0116	.8299	.0323	.7491
Rank 11 RN-Lin-Cos [198]	m_c	.0387	.5312	.0771	.2716	.0577	.2364	.0783	.0916	.1533	.4280	.1770	.2827	.1324	.0978	.1336	.0763
	m_a	.2883	.1581	.2907	.1570	.2817	.1805	.2850	.1292	.3019	.1723	.2978	.1080	.2688	.0557	.2726	.0801
	m_l	.0168	.0972	.0121	.0561	.0169	.0895	.0138	.0609	.0812	.0200	.0239	.0403	.0116	.2026	.0156	.1378
	m_s	.1721	−.0552	.1904	−.0890	.0984	.1850	.1653	−.0656	.1777	−.0729	.1538	−.0606	.0246	.2373	.0628	.0521
Rank 12 DGK [236]	m_c	.0399	.5029	.0783	.2500	.0583	.2207	.0803	.0448	.1564	.4041	.1804	.2541	.1358	.0489	.1345	.0630
	m_a	.2891	.1536	.2924	.1467	.2837	.1690	.2862	.1217	.3052	.1537	.3003	.0930	.2716	.0357	.2754	.0612
	m_l	.0175	.0246	.0126	−.0134	.0177	.0047	.0140	.0294	.0811	.0203	.0243	.0018	.0128	.0029	.0185	−.3883
	m_s	.1756	−.0982	.1928	−.1171	.1077	.0236	.1676	−.0953	.1807	−.1094	.1550	−.0771	.0286	−.0846	.0682	−.1187

Table 2.1. Estimation accuracy of the two most accurate kernels and the state-of-the-art kernels. We report Root-Mean-Square Error (RMSE) and the coefficient of determination (R^2) of estimation of four aesthetic metrics on eight layout methods.



(a) Layout computation and estimation time.



(b) Aesthetic metric computation and estimation time.

Figure 2.4. Computation time results in log scale. The plots show estimation times for our RW-LOG-LAPLICAN kernel, which has the highest accuracy. The plot on the top shows layout computation time while the plot on the bottom shows aesthetic metric computation time. As the number of nodes increases, the gap between our estimation and layout methods enlarges in both layout time and aesthetic metric computation time. Some layout methods could not be ran on all graphs due to computation time and memory limitation of the implementation. Treemap and Gosper overlap in the layout plot because the majority of the computation time is spent on hierarchical clustering.

are ranked in the bottom three of all kernel methods we tested.

A possible explanation for this is that certain types of graphlets are essential for accurate estimation. The kernels using RW sampling, which samples connected graphlets very efficiently, show a higher accuracy than other kernels. While disconnected graphlets are shown to be essential for classification problems in bioinformatics [198], for our problem, we suspect that connected graphlets are more important for accurate estimation.

Other sampling methods are not suitable for sampling connected graphlets. There are 49 possible graphlets, where the size of each graphlet is $k \in \{3, 4, 5\}$, and 29 of them are connected graphlets (Figure 2.1). However, when we sample 10,000 graphlets per graph using RN sampling, on average only 1.913% ($SD = 6.271$) of sampled graphlets are connected graphlets. Furthermore, 35.77% of graphs have no connected graphlets in the samples even though all graphs are connected, making it impossible to effectively compare graphs.

The kernels using LOG scaling show more accurate estimations than the ones using LIN. We suspect this is because of the distribution of graphlets, which often exhibit a power-law distribution. Thus, when using LOG scaling, a regression model is less affected by the graphlets with overwhelming frequencies and becomes better at discriminating graphs with different structures.

Our estimation times are fast and scale well, as shown in Figure 2.4. At around 200 nodes, our estimation times become faster than all other layout computation times. Our estimation times also outperform the computations of the four aesthetic metrics past 1,000 nodes. As the size of a graph increases, the differences become larger, to the point that our RW-LOG-LAPLACIAN takes several orders of magnitude less time than both layout computation and metric computation. Normally the layout has to be calculated in order to calculate aesthetic metrics. This is not the case for our estimation as the aesthetic metrics can be estimated without the layout result, leading to a considerable speed up.

It is interesting to see that each sampling method shows different computation

times, even though we sampled the same amount of graphlets. A possible explanation for this discrepancy can be attributed to locality of reference. We stored the graphs using an adjacency list data structure in memory. Thus, RW sampling, which finds the next sample nodes within the neighbors of currently sampled nodes, tends to exhibit good locality. On the other hand, RN sampling chooses nodes randomly, thus it would show poor locality which leads to cache misses and worse performance. The sampling method of DGK is a variant of RN. After one graphlet is randomly sampled, its immediate neighbors are also sampled. This would have better locality than RN and could explain the better computation time.

In terms of training, except for DGK, all other kernels spend negligible times computing the kernel matrix, with an average of 5.99 seconds ($SD = 3.26$). However, computing the kernel matrix of DGK takes a considerable amount of time because it requires computation of language modeling (implemented by [191]). On average it take 182.96 seconds ($SD = 9.31$).

Since there are many parameters of each layout method, and most parameters are not discrete, it is impossible to test all combinations of parameter settings. To simplify the study, we only used default parameters for each layout method. It is possible to apply our approach to different predefined parameter settings on the same layout method. However, generating new training data for multiple settings can be time consuming.

2.4 Evaluation 2: What Would a Graph Look Like in This Layout?

In this section, we describe our user study that evaluates how well our WGL method (Section 2.2.2) is able to find graphs that users assess as being perceptually similar to the actual layout results.

2.4.1 Experimental Design

We designed a ranking experiment to compare topological similarity ranks (r_{T}) obtained by our WGL method and perceptual similarity ranks (r_{P}) assessed by humans. That is, if both our WGL method and participants' choices match, then we can conclude our

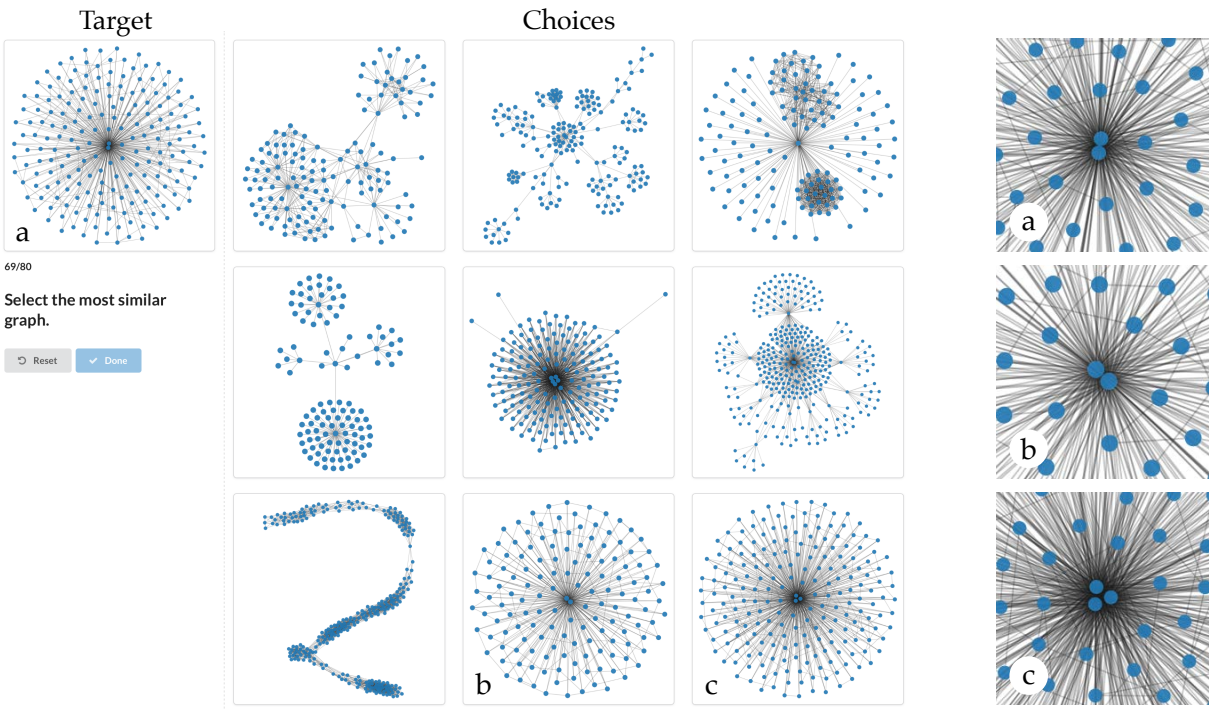


Figure 2.5. A task from the user study. For each task, the participants were given one target graph and nine choice graphs. They were then asked to rank the three most similar graphs in order of decreasing similarity. The three images on the right show the central nodes of (a), (b), and (c).

WGL method is able to find perceptually similar graphs to the actual layout results.

2.4.1.1 Task

For each task, participants were given one target graph and nine choice graphs. An example of a task given to participants is shown in Figure 2.5. They were asked to rank the three most similar graphs to the target graph in order of decreasing similarity. The instructions were given as such, "select the most similar graph." for the first choice and "select the next most similar graph." for the second and third choice. To avoid biases, we did not specify what is "similar" and let the participants decide for themselves. In each task, the same layout method was used for target and choice graphs. We did not notify this to the participants. To test our approach using graphs with different topological structures and different layout methods, we used 72 tasks comprised of nine graphs and eight layout methods.

While other task designs are possible, such as participants ranking all nine graphs or individually rating all nine choices, during our pilot study we found that these task designs are overwhelming to the participants. The participants found it very difficult to rank between dissimilar graphs. For example, in Figure 2.5, selecting the fourth or fifth most similar graph is challenging as there is little similarity to the target graph after the first three choices. Also, the focus of our evaluation is to determine how well our WGL method is able to find similar graphs, not dissimilar graphs. Thus, we only need to see which graphs our participants perceive as similar in the task, which simultaneously reduces task load when compared to ranking all nine graphs.

2.4.1.2 Graphs

To gather an unbiased evaluation, the target graphs and their choice graphs must be carefully selected.

Selecting target graphs. To test our approach on graphs with different topological structures, we select nine target graphs as follows: We clustered the 8,263 graphs into nine groups using spectral clustering [199]. For each cluster, we select ten representative graphs which have the highest sum of topological similarity within the cluster (i.e., the ten nearest graphs to the center of each cluster). Then, we randomly select one graph from these ten representative graphs to be the target graph. Figure 2.3 shows the selected nine target graphs in FM^3 layout [98].

Selecting choice graphs. To test our approach on graphs with different topological similarities obtained by graph kernels, we select nine choice graphs for each target graph as follows: We compute nine clusters of graphs based on the topological similarity between a graph and the target graph using Jenks natural breaks [124], which can be seen as one dimensional k-means clustering. We designate one of the nine choice graphs to represent what our approach would predict. This is done by selecting the graph with the highest similarity to the target graph from the cluster nearest to the target graph. For the other eight clusters, we randomly select one choice graph from a set of ten graphs that have the highest sum of similarity within the cluster (i.e., the ten nearest graphs to the center of each cluster). These can be considered as representative graphs of each

cluster. A list of the selected nine choice graphs for each target graph can be found in [9].

When we select a choice graph, we filter out graphs that have less than half or more than double the amount of nodes as the target graph. This can be considered as a constraint in our WGL method Section 2.2.2. To evaluate under the condition that isomorphic graphs to the target graph are not present in the training data, we also filter out graphs that have the same number of nodes and edges as the target graph.

The clustering approach based on topological similarity can be used for defining a topological classification of graphs. However, it would require further analysis of resultant clusters.

2.4.2 Apparatus

We used our RWS-LOG-LAPLACIAN kernel to compute the topological similarity between graphs. Although other kernels can be used, we decided on one that shows the highest estimated accuracy in the first evaluation and provides the best chance to see if a kernel can find perceptually similar graphs.

The experiment was conducted on an iPad Pro which has a 12.9 inch display with 2,732×2,048 pixels. Each of the graphs were presented in 640×640 pixels. All nodes were drawn using the same blue color (Red: .122, Green: .467, Blue: .706, Alpha: .9) and edges were drawn using dark grey color (Red: .1, Green: .1, Blue: .1, Alpha: .25), as shown in Figure 2.5.

2.4.3 Procedure

Prior to the beginning of the experiment, the participants were asked several questions about demographic information, such as age and experience with graph visualization. To familiarize participants with the system, eight training tasks were given, at which time they were allowed to ask any questions to the moderator. Once the training was done, the moderator did not communicate with the participant. The 72 tasks were presented in randomized order to the participants. The nine choice graphs of each task were also presented in a randomized order. For each task, we asked the participants to

briefly explain why he or she selected the particular graph (think-aloud protocol).

2.4.4 Participants

We recruited 30 (9 females and 21 males) participants in our user study. The ages of the participants ranged from 18 to 36 years, with the mean age of 26.67 years (SD = 3.68). Most of the participants were from science and engineering backgrounds: 22 computer science, 2 electrical and computer engineering, 2 cognitive science, 1 animal science, 1 political science, and 1 literature. 28 participants indicated that they had seen a graph visualization (e.g., a node-link diagram), 21 participants had used one, and 16 participants had created one before. On average, each participant took 28.94 minutes ($SD = 11.49$) to complete all 72 tasks.

2.4.5 Results

For each task, the nine choices receive a topological similarity rank (r_{T}) from one to nine in order of decreasing topological similarity. We define predicted choice as the choice graph that our method ranks as the most topologically similar to the target graph ($r_{\mathrm{T}} = 1$). Based on the responses by the participants, the three choices receive a perceptual similarity rank (r_{P}) from one to three, where one is the first chosen graph. Choices not selected by the participants are ranked $r_{\mathrm{P}} = 4$. Results of our evaluation can be found in Figure 2.6.

Overall, 80.46% of the time, participants' responses for rank one ($r_{\mathrm{P}} = 1$) match our predicted choices ($r_{\mathrm{T}} = 1$). 90.27% of participants' responses within rank one and rank two ($r_{\mathrm{P}} = \{1, 2\}$) contain the predicted choices, and 93.80% of responses within ranks one, two, and three ($r_{\mathrm{P}} = \{1, 2, 3\}$) contain the predicted choice. A Friedman test (non-parametric alternative to one-way repeated measures ANOVA) shows a significant effect of the topological similarity rankings (r_{T}) on the perceptual similarity rankings (r_{P}) with $\chi^2(8) = 6343.9,\ p < .0001$. The mean r_{P} of the predicted choices is 1.35 ($SD = .82$ and $IQR = 0$), which is clearly different than other choices ($r_{\mathrm{T}} > 1$) as shown in Table 2.2. Post-hoc analysis with Wilcoxon signed-rank tests using Bonferroni correction confirms that the predicted choices are ranked significantly higher ($p < .0001$)

	1	2	3	4	5	6	7	8	9
Mean	**1.35**	3.36	3.45	3.68	3.31	3.45	3.82	3.74	3.83
SD	.82	.96	.84	.71	.91	.84	.53	.63	.53
Median	**1**	4	4	4	4	4	4	4	4
IQR	0	1	1	0	1	1	0	0	0

Table 2.2. Descriptive statistics of perceptual similarity rank (r_P) for each topological similarity rank (r_T). Our predicted choices are ranked on average 1.35 by the participants.

than all other choices.

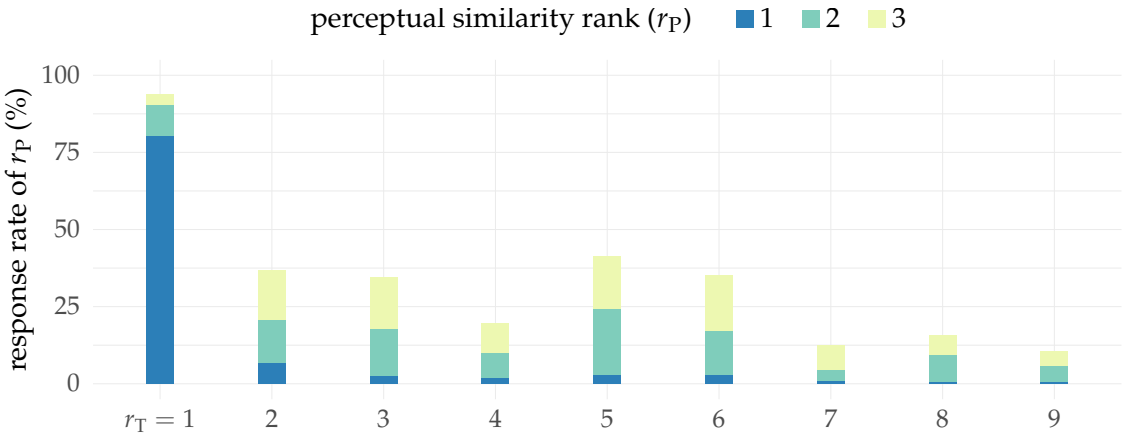


Figure 2.6. Response rate of perceptual similarity rank (r_P) for each topological similarity rank (r_T).

To see the effect layout methods have on participants' responses, we break down the responses with a topological similarity rank of one ($r_T = 1$), or predicted choice, by each layout method (Figure 2.7). Except for Spectral and Treemap, the predicted choices are ranked in more than 78.52% (up to 93.33%) of participants' responses as being the most perceptually similar. In more than 94.44% (up to 99.26%) of participants' responses, the predicted choices are within the three most perceptually similar graphs ($r_P = \{1, 2, 3\}$), as shown in Figure 2.6b. For Spectral, the predicted choices are ranked in 72.22% of participants' responses as being the most similar graph, and 84.07% of responses as being within the three most similar graphs. For Treemap, the predicted choices are ranked in 59.63% of participants' responses as the most similar graph, and 82.59% of responses as being within the three most similar graphs. A Friedman test shows a significant effect of layout method on perceptual similarity rankings (r_P) with

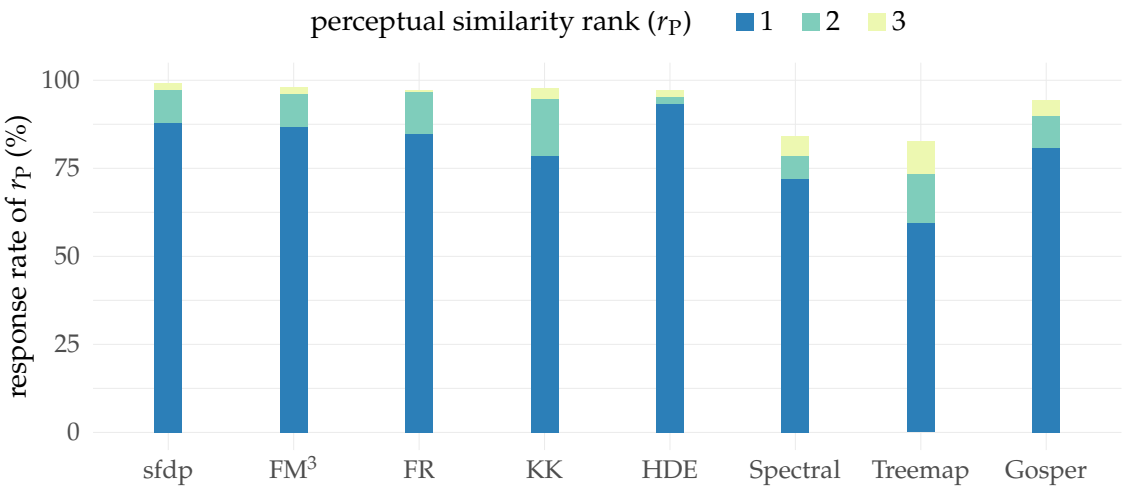


Figure 2.7. Response rate on topological similarity rank of 1 ($r_T = 1$) for each layout method.

	sfdp	FM^3	FR	KK	HDE	Spct.	Tree.	Gos.
Mean	1.16	1.19	1.21	1.29	1.14	1.65	1.84	1.35
SD	.46	.55	.6	.63	.58	1.14	1.17	.81
Median	1	1	1	1	1	1	1	1
IQR	0	0	0	0	0	1	2	0

Table 2.3. Descriptive statistics of perceptual similarity rank (r_P) on topological similarity rank of 1 ($r_T = 1$) for each layout method.

$\chi^2(7) = 200.85$, $p < .0001$. Except for Spectral and Treemap, the mean r_P of the predicted choices for each layout method is close to 1, from 1.16 to 1.35 ($SD = .46$–$.81$ and $IQR = 0$), as shown in Table 2.3. The means r_P of Spectral and Treemap are 1.65 ($SD = 1.14$ and $IQR = 1$) and 1.84 ($SD = 1.17$ and $IQR = 2$), respectively. Post-hoc analysis shows that the predicted choices with Treemap are ranked significantly lower than the predicted choices with other layout methods ($p < .0001$) except for Spectral. The predicted choices with Spectral are also ranked by participants as being significantly lower than the predicted choices with other layout methods ($p < .05$) except for Gosper and Treemap.

We also break down the responses for the topological similarity rank of one ($r_T = 1$) by each target graph (Figure 2.8). Except for G_{2331} and G_{3833}, the predicted choices are ranked in more than 79.58% (up to 98.33%) of responses as being the most similar, and more than 92.08% (up to 99.99%) of responses as being within the three most similar

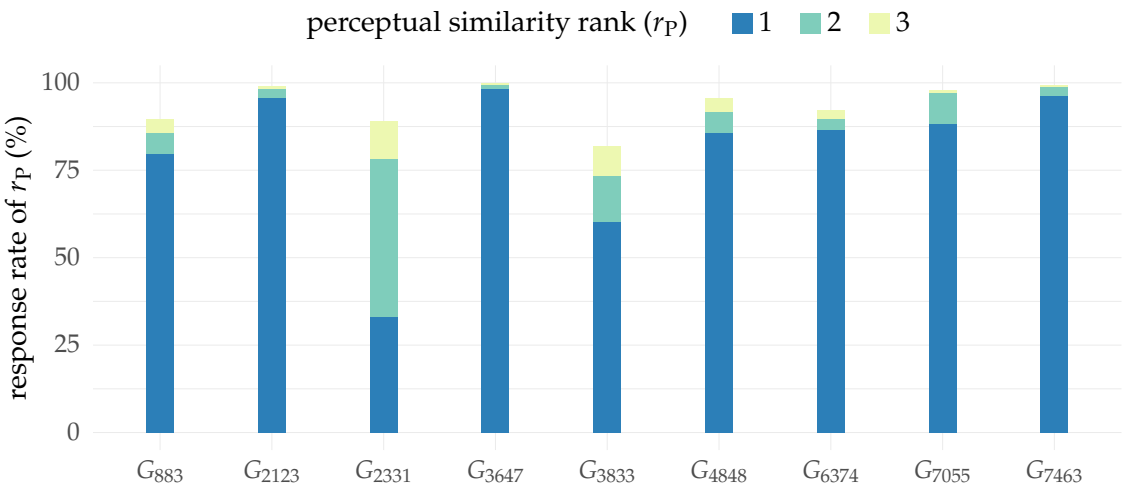


Figure 2.8. Response rate on topological similarity rank of 1 ($r_T = 1$) for each target graph.

	G_{883}	G_{2123}	G_{2331}	G_{3647}	G_{3833}	G_{4848}	G_{6374}	G_{7055}	G_{7463}
Mean	1.45	1.07	2	1.02	1.85	1.27	1.32	1.17	1.06
SD	.98	.36	.94	.17	1.18	.74	.86	.53	.34
Median	1	1	2	1	1	1	1	1	1
IQR	0	0	1	0	2	0	0	0	0

Table 2.4. Descriptive statistics of perceptual similarity rank (r_P) on topological similarity rank of 1 ($r_T = 1$) for each target graph.

graphs ($r_P = \{1, 2, 3\}$), as shown in Figure 2.6c. For G_{2331}, the predicted choices are ranked in 32.92% of all responses as being the most similar graph, 78.33% of responses as being within the two most similar graphs, and 89.16% of responses as being within the three most similar graphs. For G_{3833}, the predicted choices are ranked in 60.42% of participants' responses as being the most similar graph, 73.33% of responses as being within the two most similar graphs, and 81.67% of responses as being within the three most similar graphs. A Friedman test shows a significant effect of target graphs on perceptual similarity rankings (r_P) with $\chi^2(8) = 511$, $p < .0001$. Except for G_{2331} and G_{3833}, the mean r_P of the predicted choices for each target graph is close to 1, from 1.06 to 1.45 ($SD = .34$–$.98$ and $IQR = 0$), as shown in Table 2.4. The means r_P of G_{2331} and G_{3833} are 2 ($SD = .94$ and $IQR = 1$) and 1.85 ($SD = 1.18$ and $IQR = 2$), respectively. Post-hoc analysis shows that the predicted choices of G_{2331} and G_{3833} are ranked significantly lower than predicted choices of other target graphs ($p < .0001$).

2.4.6 Discussion

The results of the user study show that in more than 80% of participants' responses, the predicted choices are ranked as being the most perceptually similar graph to the target graphs. Also, more than 93% of the responses ranked the predicted choices as being within the three most perceptually similar graphs. Thus, we believe our WGL method is able to provide the expected layout results that are perceptually similar to the actual layout results.

When we analyze participants' responses for the predicted choices ($r_{\mathrm{T}} = 1$) for each layout separately, we find that the predicted choices with Spectral and Treemap layouts are ranked lower than with other layouts. The common reasons given by participants for selecting choice graphs with Spectral layout were "line shape" and "number of nodes". We notice that the Spectral layouts have many nodes that overlap each other. Treemap, on the other hand, produces similar looking graphs due to its geometric constraints. This observation was mirrored by many participants who said "they all look similar" for the choices with Treemap layout. Common reasons for selecting choice graphs with Treemap layout were "edge density" and "overall edge direction".

It is interesting to see how people perceive certain structures as more important than others. For instance, when we look at the responses on target graphs separately, we notice that target graph G_{2331} has different response patterns. Target graph G_{2331} and its choice graph are shown in Figure 2.5. Participants' responses for $r_{\mathrm{P}} = 1$ are split between two choices, Figure 2.5b and Figure 2.5c. The common reasons why the participants ranked Figure 2.5c as the most similar to Figure 2.5a were "density", "shape", and "number of edges". On the other hand, the common reason why the participants ranked Figure 2.5b as the most similar to Figure 2.5a was "the number of central nodes". Our method also chose Figure 2.5c as the most similar graph because of the general structure matching the target graph, but ranked Figure 2.5b as being second most similar $r_{\mathrm{T}} = 2$. In the case of target graph G_{2331}, the number of nodes in the center held more value to some participants than the overall structure. For the sake of the study, only one similar graph was chosen by our system per target graph. In a real

system, the user would be given several similar looking graphs, including isomorphic graphs. Thus, the real system would have a higher chance of showing the accurate expected layout results than in the setting of our user study.

2.5 Related Work

In this section, we discuss other related works that are not mentioned in this chapter. Only a handful of studies have used topological features for visualizing graphs. Perhaps this is why there is a scarcity of studies applying machine learning techniques to the process of visualizing a graph [69].

Niggemann and Stein [172] introduced a learning method to find an optimal layout method for a clustered graph. The method constructs a handcrafted feature vector of a cluster from a number of graph measures, including the number of nodes, diameter, and maximum node degree. Then, it attempts to find an optimal layout for each cluster. However, these features have been proved as not expressive enough to capture topological similarities in many graph kernel works [173].

Behrisch et al. [23] proposed a technique called Magnostics, where a graph is represented as a matrix view and image-based features are used to find similar matrices. One of the challenges of a matrix view is the node ordering. Depending on the ordering, even the same graph can be measured as a different graph from itself. Graph kernels do not suffer from the same problem since they measure the similarity using only the topology of the graph.

Several techniques have used machine learning approaches to improve the quality of a graph layout [69]. Some of these techniques used evolutionary algorithms for learning user preferences with a human-in-the-loop assessment [14, 19, 155, 204], while others have designed neural network algorithms to optimize a layout for certain aesthetic criteria [50, 162, 221]. One major limitation of these techniques is that models learned from one graph are not usable in other graphs. Since these techniques often require multiple computations of layouts and their aesthetic metrics, the learning process can be highly time-consuming. These techniques can benefit from our approach by quickly

showing the expected layout results and estimating the aesthetic metrics.

Many empirical studies have been conducted to understand the relation between topological characteristics of graphs and layout methods. The main idea is to find the "best" way to lay out given graphs [87]. To achieve this, Archambault et al. [12] introduced a layout method which first recursively detects the topological features of subgraphs, such as whether a subgraph is a tree, cluster, or complete graph. Then, each subgraph is laid out using the suitable method according to its topological characteristics. A drawback of this method is that the feature detectors are limited to five classes. Our kernel can be utilized for heterogeneous feature detection with less computational cost.

A number of recent studies investigated sampling methods for large graph visualization. Wu et al. [232] evaluated a number of graph sampling methods in terms of resulting visualization. They found that different visual features were preserved when different sampling strategies were used. Nguyen et al. [171] proposed a new family of quality metrics for large graphs based on a sampled graph.

2.6 Conclusion

We have developed a machine learning approach using graph kernels for the purpose of showing what a graph would look like in different layouts and their corresponding aesthetic metrics. We have also introduced a framework for designing graphlet kernels, which allows us to derive several new ones. The estimations using our new kernels can be derived several orders of magnitude faster than computing the actual layouts and their aesthetic metrics. Also, our kernels outperform state-of-the-art kernels in both accuracy and computation time. The results of our user study show that the topological similarity computed with our kernel matches perceptual similarity assessed by human participants.

Chapter 3

Augmenting the Design Process of Node-Link Diagrams using a Deep Generative Model

Finding a good layout of a graph is a challenging task. The heuristics to find a good layout are nearly impossible to define due to the vast design space. It requires to consider many different graphs, characteristics to be highlighted, and user preferences. There is thus no existing method to automatically find a good layout. In practice, users rely on a trial-and-error process to find a good layout. Until they find a layout that satisfies their requirements (e.g., highlighting the community structure of a graph), users typically visualize a graph in multiple layouts using different methods and varying parameter settings. This process often requires a significant amount of the user's time as it results in a haphazard and tedious exploration of a large number of layouts [29].

Furthermore, expert knowledge of layout methods is often required to find a good layout. Most layout methods have a number of parameters that can be tweaked to improve the layout of a graph. However, many layout methods—especially force-directed ones—are very sensitive to the parameter values [169], where the resulting layouts can be incomprehensible or even misleading [70]. A proper selection of the parameter settings for a given graph requires detailed knowledge of the chosen method. Such knowledge can only be acquired through extensive experience in graph visualization.

Thus, novice users are often blindly tweaking parameters, which leads to many trials and errors as they cannot foresee what the resulting layout will look like. Moreover, novices might explore only a fraction of possible layouts, choose an inadequate layout, and thus overlook critical insights in the graph.

To help users to produce a layout that best suits their requirements, we present a deep generative model that systematically visualizes a graph in diverse layouts. We design an encoder-decoder architecture to learn a generative model from a collection of example layouts, where the encoder represents training examples in a latent space and the decoder generates layouts from the latent space. In particular, we train the model to construct a two-dimensional latent space. By mapping a grid of generated samples, a two-dimensional latent space can be used as a what-you-see-is-what-you-get (WYSIWYG) interface. This allows users to intuitively navigate and generate various layouts without blindly tweaking parameters of layout methods. Thus, users can create a layout that satisfies their requirements without a haphazard trial-and-error process or any expert knowledge of layout methods.

The results of our evaluations show that our model is capable of learning and generalizing abstract concepts of graph layouts, not just memorizing the training examples. Also, graph neural networks [135, 234] and Gromov–Wasserstein distance [160] help the model better learn the complex relationship between the structure and the layouts of a graph. After training, our model generates new layouts considerably faster than existing layout methods. In addition, the generated layouts are spatially stable, which helps users to compare various layouts.

In summary, this chapter introduces a fundamentally new approach to graph visualization, where a machine learning model learns to visualize a graph as a node-link diagram from existing examples without manually-defined heuristics. This approach is an example of *artificial intelligence augmentation* [44]: a machine learning model builds a new type of user interface (i.e., layout latent space) to augment a human intelligence task (i.e., graph visualization design).

3.1 Related Work

Our work is related to graph visualization, deep generative modeling, and deep learning on graphs. We discuss related work in this section.

3.1.1 Graph Visualization

A plethora of layout methods has been introduced over the last five decades. In this chapter, we focus on two-dimensional layout methods that produce straight-edge drawings, such as force-directed methods [60,71,81,83,127], dimensionality reduction-based methods [35,107,142], spectral methods [51,140], and multi-level methods [82,85, 98,106,116,219]. Many analysts do not have expert knowledge in these layout methods for finding a good layout. Our goal is to learn the layouts produced by the different methods in a single model. In addition, we provide an intuitive way for users to explore and generate diverse layouts of a graph without the need of expert knowledge of multiple layout methods.

To help users find a good layout, several methods have been developed to accelerate the trial-and-error selection process by learning user preferences [19,29,155,204]. For example, Biedl et al. [29] have introduced the concept of *multidrawing*, which systematically produces many different layouts of the same graph. Some methods use an evolutionary algorithm [19,155,204] to optimize a layout based on a human-in-the-loop assessment. However, these methods require constant human intervention throughout the optimization process. In addition, the goal of their optimization is to narrow down the search space. This allows the model to create only a limited number of layouts. Hence, multiple learning sessions might be needed to allow users to investigate other possible layouts of the same graph. In contrast to these models, our approach is to train a machine learning model in a fully unsupervised manner to produce diverse layouts, not to narrow the search space.

Recently, several machine learning approaches have been introduced to different tasks in graph visualization, such as previewing large graphs [145], exploring large graphs [47], and evaluating visualizations [103,136]. Unlike these approaches, our goal is to train a model to generate layouts.

3.1.2 Deep Generative Models

The term "generative model" can be used in different ways. In this chapter, we refer to a model that can be trained on unlabeled data and is capable of generating new samples that are similar to, but not the same as, the training data. For example, we can train a model to create synthetic images of handwritten digits by learning from a large collection of real ones [92, 134]. Since generating new, realistic samples requires a good understanding of the given data, generative modeling is often considered as a key component of unsupervised learning.

In recent years, generative models built with deep neural networks and stochastic optimization methods have demonstrated state-of-the-art performance in various applications, such as text generation [117], music generation [193], and drug design [90]. While several approaches have been proposed for deep generative modeling, the two most prominent ones are variational autoencoders (VAEs) [134] and generative adversarial networks (GANs) [92].

VAEs and GANs have their own advantages and disadvantages. GANs generally produce visually sharper results when applied to an image dataset as they can implicitly model a complex distribution. However, training GANs is difficult due to non-convergence and mode collapse [91]. VAEs are easier to train and provide both a generative model and an inference model. However, they tend to produce blurry results when applied to images.

For designing our generative model, we use sliced-Wasserstein autoencoders (SWAEs) [139]. As a variant of VAE, it is easier to train than GANs. In addition, it is capable of learning complex distributions. SWAEs allow us to shape the distribution of the latent space into any samplable probability distribution without training an adversarial network or defining a likelihood function.

3.1.3 Deep Learning on Graphs

Machine learning approaches to graph-structured data, such as social networks and biological networks, require an effective representation of the graph structure. Recently, many graph neural networks (GNNs) have been proposed for representation learning

on graphs, such as graph convolutional networks [135], GraphSAGE [105], and graph isomorphism networks [234]. They have achieved state-of-the-art performance for many tasks, such as graph classification, node classification, and link prediction. We also use GNNs for learning the complex relationship between the structure and the layouts of a graph.

Several generative models have been introduced for graph-structured data using GNNs [63, 95, 153, 201, 238]. These models learn to generate whole graphs for tasks that require new samples of graphs, such as *de novo* drug design. However, generating a graph layout is a different type of task, where the structure of a graph remains the same, but the node attributes (i.e., positions) are different. Thus, we need a model that learns to generate different node attributes of the same graph.

3.2 Approach

Our goal is to learn a generative model that can systematically visualize a graph in diverse layouts from a collection of example layouts. We describe the entire process for building a deep generative model for graph layout, from collecting training data to designing our architecture.

3.2.1 Training Data Collection

Learning a generative model using deep neural networks requires a large amount of training data. As a data-driven approach, the quality of the training dataset is crucial to build an effective model. For our goal, we need a large and diverse collection of layouts of the given graph.

Grid search is often used for parameter optimization [27], where a set of values is selected for each parameter, and the model is evaluated for each combination of the selected sets of values. It is often used for producing multiple layouts of a graph. For example, Haleem et al. [103] have used grid search for producing their training dataset. However, the number of combinations of parameters increases exponentially with each additional parameter. In addition, different sets of parameter values are often required for different graphs. Therefore, it requires expert knowledge of each layout method to

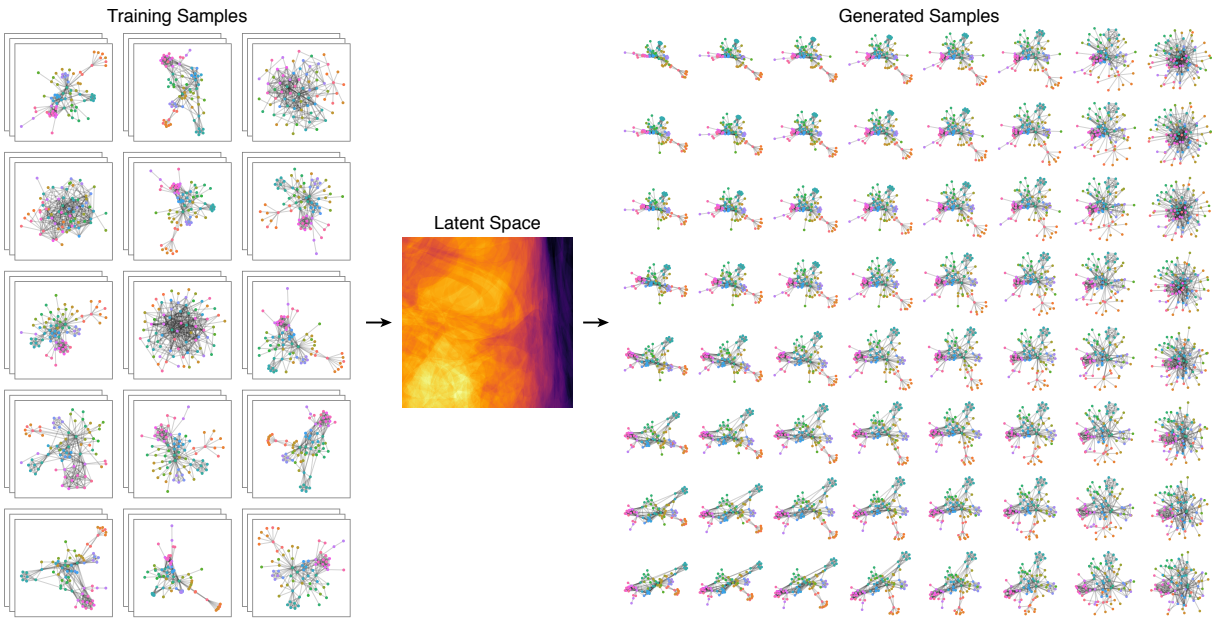


Figure 3.1. Generative modeling of layouts for the *Les Misérables* character co-occurrence network [137]. We train our generative model to construct a 2D latent space by learning from a collection of example layouts (training samples). From the grid of generated samples, we can see the smooth transitions between the different layouts. This shows that our model is capable of learning and generalizing abstract concepts of graph layouts, not just memorizing the training samples. Users can use this sample grid of the latent space as a WYSIWYG interface to generate a layout they want, without either blindly tweaking parameters of layout methods or requiring expert knowledge of layout methods. The color mapping of the latent space represents the shape-based metric [72] of the generated samples. Throughout this chapter, unless otherwise specified, the node color represents the hierarchical community structure of the graph [175, 207], so readers can easily compare node locations in different layouts. An interactive demo is available in the supplementary material [10]

carefully define the search space for collecting the training data in a reasonable amount of time.

We collect training example layouts using multiple layout methods following random search, where each layout is computed using randomly assigned parameter values of a method. We uniformly sample a value from a finite interval for a numerical parameter or a set of possible values for a categorical parameter. Random search often outperforms grid search for parameter optimization [27], especially when only few parameters affect the final result. Because the effect of the same parameter value can

vary greatly depending on the structure of the graph, we believe random search would produce a more diverse set of layouts than grid search. Moreover, for this approach, we only need to define the interval of values for a numeric parameter, which is a simpler task for non-experts than selecting specific values for grid search.

Computing a large number of layouts would take a considerable amount of time. However, we can start training a model without having the full training dataset, thanks to stochastic optimization methods. For example, we can train a model with stochastic gradient descent [192], where a small batch of training examples (typically a few dozens) are fed to the model at each step. Thus, we can incrementally train the model while generating the training examples simultaneously. This allows the users to use our model as early as possible.

3.2.2 Layout Features

Selecting informative, discriminating, and independent features of the input data is also an essential step for building an effective machine learning model. With deep neural networks, we can use low-level features of the input data without handcrafted feature engineering. For example, the red, blue, and green channel values of pixels often are directly used as the input feature of an image in deep learning models.

However, what would be a good feature of a graph layout? Although the node positions can be a low-level feature of a layout, using the raw positions as a feature has several issues.

Many graph layout methods do not use spatial position to directly encode attribute values of either nodes or edges. The methods are designed to optimize a layout following certain heuristics, such as minimizing the difference between the Euclidean and graph-theoretic distances, reducing edge crossings, and minimizing node overlaps. Therefore, the position of a node is often a side effect of the layout method; it does not directly encode any attributes or structural properties of a node [169].

In addition, many layout methods are nondeterministic since they employ randomness in the layout process to avoid local minima, such as randomly initializing the positions of nodes [169]. Therefore, the position of a node can significantly vary

between different runs of the same layout method with the same parameter setting. For these reasons, the node positions are not a reliable feature of a layout.

Besides, due to the Gestalt principle of proximity [228], the nodes placed close to each other would be perceived as a group whether or not this relationship exists [157]. For example, some nodes might be placed near each other because they are pushed away from elsewhere, not because they are closely connected in the graph [169]. However, the viewers can perceive them as a cluster because spatial proximity strongly influences how the viewer perceives the relationships in the graph [87]. Thus, spatial proximity is an essential feature of a layout.

We use the pairwise Euclidean distance of nodes in a layout as the feature of the layout. As the spatial proximity of nodes is an intrinsic feature of a layout, we can directly use the pairwise distance matrices to compare different layouts, without considering the rotation or reflection of the points. Furthermore, we can normalize a pairwise distance matrix of nodes by its mean value for comparing layouts in different scales.

Therefore, for the positions of the nodes (P) of a given layout, we compute the feature of the given layout by $X_L = D/\bar{D}$, where D is the pairwise distance matrix of P and $\bar{D}$ is the mean of D. Each row of X_L is the node-level feature of a layout for each node. Other variations of the pairwise distance can be used as a feature, such as the Gaussian kernel from the pairwise distance: $\exp\left(-D^2/2\sigma\right)$.

3.2.3 Structural Equivalence

Two nodes of a graph are said to be structurally equivalent if they have the same set of neighbors. Structurally equivalent nodes (SENs) of a graph are often placed at different locations, as many layout methods have a procedure to prevent overlapping. For example, force-directed layout methods apply repulsive forces between all the nodes of a graph. As the layout results are often nondeterministic, the positions of SENs are mainly determined by the randomness in the layout method.

For instance, Figure 3.2.3 shows two different layout results of the same graph using the same layout method (sfdp [116]) with the same default parameter setting. The blue nodes are the same set of SENs. However, the arrangements of the blue nodes between

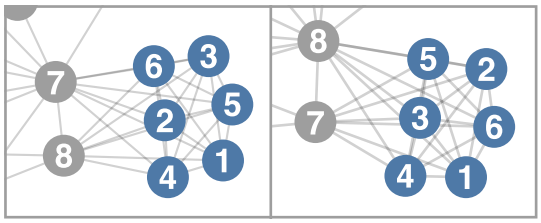


Figure 3.2. Structurally equivalent nodes of a graph are often placed at different locations, as many layout methods have a procedure to prevent overlapping.

the two layouts are quite different. In the top layout, the nodes $\{6, 2, 4\}$ are closer to $\{7, 8\}$ than $\{3, 5, 1\}$. In contrast, in the bottom layout, the nodes $\{5, 3, 4\}$ are closer to $\{7, 8\}$ than $\{2, 6, 1\}$. This presents a challenge because a permutation invariant measure of similarity is needed for the same set of SENs. In other words, we need a method to permute the same set of SENs, from one possible arrangement to the other, for a visually correct similarity measure.

To address this issue, we compare two different layouts of the same set of SENs using the Gromov–Wasserstein (GW) distance [160, 178]:

$$\mathrm{GW}(C, C') = \min_{T} \sum_{i,j,k,l} L\left(C_{i,k}, C'_{j,l}\right) \; T_{i,j} \; T_{k,l}, \tag{3.1}$$

where C and C' are cost matrices representing either similarities or distances between the objects of each metric space, L is a loss function that measures the discrepancy between the two cost matrices (e.g., L_2 loss), and T is a permutation matrix, which couples the two metric spaces, that minimizes L. The GW distance measures the difference between two metric spaces. For example, it can measure the dissimilarity between two point clouds, invariant to the permutations of the points.

For a more efficient computation, we do not backpropagate through the GW distance in the optimization process of our model. Instead, we use the permutation matrix T to permute the same set of SENs. We describe this in more detail in Section 3.2.5

3.2.4 Architecture

We design an encoder-decoder architecture that learns a generative model for graph layout. Our architecture and optimization process generally follow the framework of

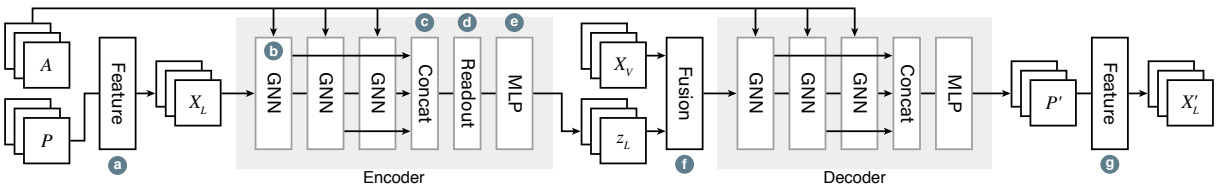


Figure 3.3. Our encoder-decoder architecture that learns a generative model from a collection of example layouts. We describe it in Section 3.2

VAEs [134]. The overview of our architecture is shown in Figure 3.3.

Preliminaries An autoencoder (AE) learns to encode a high-dimensional input object to a lower-dimensional latent space and then decodes the latent representation to reconstruct the input object. A classical AE is typically trained to minimize *reconstruction loss*, which measures the dissimilarity between the input object and the reconstructed input object. By training an AE to learn significantly lower-dimensional representations than the original dimensionality of the input objects, the model is encouraged to produce highly compressed representations that capture the essence of the input objects.

A classical AE does not have any regularization of the latent representation. This leads to an arbitrary distribution of the latent space, which makes it difficult to understand the shape of the latent space. Therefore, if we decode some area of the latent space, we would get reconstructed objects that do not look like any of the input objects, as that area has not been trained for reconstructing any input objects.

VAEs [134] extend the classical AEs by minimizing *variational loss*, which measures the difference of the distribution of the input objects' latent representations and a *prior* distribution. Minimizing the variational loss encourages VAEs to learn the latent space that follows a predefined structure (i.e., prior distribution). This allows us to know which part of the latent space is trained with some input objects. Thus, it is easy to generate a new object similar to some of the input objects.

Encoder For our problem, the input objects are graph layouts, i.e., the positions of nodes (P, each row is the position of a node). We first compute the feature of a layout X_L as discussed in Section 3.2.2 (Figure 3.3a), where each row corresponds to the input feature of a node.

The encoder takes the feature of a layout X_L and the structure of a graph (A, the adjacency matrix), as shown in Figure 3.3b. Then, it outputs the latent representation of a layout z_L. We use graph neural networks (GNNs) to take the graph structure into account in the learning process.

In general, GNNs learn the representation of a node following a recursive neighborhood aggregation (or message passing) scheme, where the representation of a node is derived by recursively aggregating and transforming the representations of its neighbors. For instance, graph isomorphism networks (GINs) [234] update node representations as:

$$h_v^{(k)} = \text{MLP}^{(k)}\left((1+\epsilon^{(k)})\cdot h_v^{(k-1)} + f^{(k)}\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)\right), \tag{3.2}$$

where $h_v^{(k)}$ is the feature vector of node v at the k-th aggregation step ($h_v^{(0)}$ is the input node feature), $\mathcal{N}(v)$ is a set of neighbors of node v, f is a function that aggregates the representations of $\mathcal{N}(v)$, such as element-wise mean, ϵ is a learnable parameter or a fixed scalar that weights the representation of node v and the aggregated representation of $\mathcal{N}(v)$, and MLP is a multi-layer perceptron to learn a function that transforms node representations. After k steps of aggregation, our encoder produces the latent representation of a node, which captures the information within the node's k-hop neighborhood.

The aggregation scheme of GNNs corresponds to the subtree structure rooted at each node, where it learns a more global representation of a graph as the number of aggregations steps increases [234]. Therefore, depending on the graph structure, an earlier step may learn a better representation of a node. To consider all representations at different aggregation steps, we use the outputs of all GNN layers. This is achieved by concatenating the outputs of all GNN layers (Figure 3.3c) similar to [235].

The graph-level representation of a layout is obtained with a readout function (Figure 3.3d). A readout function should be permutation invariant to learn the same graph-level representation of a graph, regardless of the ordering of the nodes. It can be a simple element-wise mean pooling or a more advanced graph-level pooling, such as DiffPool [237]. We use MLP to produce the final output representation of a layout z_L

(Figure 3.3e).

Although we could use a higher-dimensional space, we construct a 2D latent space since users can intuitively navigate the latent space. By mapping a grid of generated samples, a 2D latent space can be used as a WYSIWYG interface (more in Section 3.4). For this, we set the prior distribution as the uniform distribution in $[-1, 1]^2$. Thus, the encoder produces a 2D vector representation of a layout z_L in $[-1, 1]^2$.

Fusion Layer The encoder produces a graph-level representation of each layout z_L. If we only use this graph-level representation, all nodes will have the same feature value for the decoder. To distinguish the individual nodes, we use one-hot encoding of the nodes (i.e., identity matrix) as the feature of the nodes X_V, similar to the featureless case of GCN [135]. Then, we combine the graph-level representation of a layout z_L and node-level features X_V using a fusion layer [121] (Figure 3.3f). It fuses z_L and X_V by concatenating each row of X_V with z_L.

Decoder The decoder takes the fused features and learns to reconstruct the input layouts, i.e., it reconstructs the position of the nodes P'. The decoder has a similar architecture to the encoder, except that it does not have a readout function, as the output of the decoder is a node-level representation, i.e., the positions of the nodes P'. The feature of the reconstructed layout X'_L is computed for measuring the reconstruction loss (Figure 3.3g). After training, users can generate diverse layouts by feeding different z_L values to the decoder.

3.2.5 Training

Following the framework of VAEs [134], we learn the parameters of the neural network used in our model by minimizing the reconstruction loss L_X and the variational loss L_Z.

The reconstruction loss L_X measures the difference between the input layout P and its reconstructed layout P'. As we have discussed in Section 3.2.2, we compare the features of the two layouts (X and X') to measure the dissimilarity between them. For example, we can use the L_1 loss function between the two layouts $L_X = \|X_L - X'_L\|_1$.

As we have discussed in Section 3.2.3, for the same set of structurally equivalent

nodes (SENs) in a graph, we use the Gromov–Wasserstein (GW) distance between the two layouts (P and P') for the comparison. However, for a more efficient computation, we do not backpropagate through the GW distance in the optimization process. Computing GW yields a permutation matrix (T in Equation 3.1), as it is based on ideas from mass transportation [216]. We use this permutation matrix to permute the input layout $\hat{P} = TP$ and compute the feature of the permuted input layout $\hat{X}_L$. Then, we compute the reconstruction loss between the permuted input layout $\hat{P}$ and the reconstructed layout P'. For example, if we use the L_1 loss, we can compute the reconstruction loss as $L_X = \|\hat{X}_L - X'_L\|_1$. With this method, we can save computation cost for the backpropagation through the complex GW computation, and we still can compare the different layouts of the SENs properly, without affecting the model performance.

In this chapter, we use the variational loss function defined in sliced-Wasserstein autoencoders (SWAE) [139]:

$$\mathrm{SW}_c(p, q) = \frac{1}{L \cdot M} \sum_{l=1}^{L} \sum_{m=1}^{M} c\Big(\theta_l \cdot p_{i[m]},\ \theta_l \cdot q_{j[m]}\Big)\,, \tag{3.3}$$

where p and q are the samples from two distributions, M is the number of the samples, θ_l are random *slices* sampled from a uniform distribution on a d-dimensional unit sphere (d is the dimension of p and q), L is the number of random slices, $i\,[m]$ and $j\,[m]$ are the indices of sorted $\theta_l \cdot p_{i[m]}$ and $\theta_l \cdot q_{j[m]}$ with respect to m, correspondingly, c is a transportation cost function (e.g., L_2 loss).

Using sliced-Wasserstein distance for variational loss allows us to shape the distribution of the latent space into any samplable probability distribution without defining a likelihood function or training an adversarial network. The variational loss of our model is $L_Z = \mathrm{SW}_c(Z_L, Z_P)$, where Z_P is a set of samples of the prior distribution.

The optimization objective can be written as:

$$\underset{E_\theta, D_\phi}{\operatorname{argmin}}\ L_X + \beta L_Z, \tag{3.4}$$

where E_θ is the encoder parameterized by θ, D_ϕ is the decoder parameterized by ϕ, and β is the relative importance of the two losses.

Name	Type	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	$\|S\|$	$\|S\|/\|V\|$	l	Source
lesmis	Co-occurrence	77	254	3.30	35	.455	2.61	[137, 144]
can96	Mesh structure	96	336	3.5	0	0	4.36	[61]
football	Interaction network	115	613	5.33	0	0	2.49	[88, 144]
rajat11	Circuit simulation	135	276	2.04	6	.044	5.57	[61]
jazz	Collaboration	198	2,742	13.85	14	.071	2.22	[89, 144]
netsci	Coauthorship	379	914	2.41	183	.483	6.03	[144, 170]
dwt419	Mesh structure	419	1,572	3.75	32	.076	8.97	[61]
asoiaf	Co-occurrence	796	2,823	3.55	170	.214	3.41	[144]
bus1138	Power system	1,138	1,458	1.28	16	.014	12.71	[61]

Table 3.1. The nine graphs used in the evaluation. $|V|$: the number of nodes, $|E|$: the number of edges, $|S|$: the number of nodes having structural equivalence, l: average path length.

3.3 Evaluation

The main goal of our evaluations is to see whether our model is capable of learning and generalizing abstract concepts of graph layouts, not just memorizing the training examples. We perform quantitative and qualitative evaluations of the reconstruction of unseen layouts (i.e., the test set), and a qualitative evaluation of the learned latent space.

3.3.1 Datasets

We use nine real-world graphs and 20,000 layouts per graph in our evaluations. For the quantitative reconstruction evaluation, as we perform 5-fold cross-validations, 16,000 layouts are used as the training set, and 4,000 layouts are used as the test set.

Graphs Table 3.1 lists the graphs used for our evaluations. These include varying sizes and types of networks. We collected the graphs from publicly available repositories [61, 144]. As disconnected components can be laid out independently, we use the largest component if a graph has multiple disconnected components (netsci). can96 and football do not have any structurally equivalent nodes. Therefore, we did not use the Gromov–Wasserstein distance [160].

Layouts We collected 20,000 layouts for each graph using the four different layout methods and 5,000 different parameter settings per method, where each parameter value

is randomly assigned following random search (Section 3.2.1). The layout methods and their parameter ranges used in the evaluations are listed in Table 3.2.

While there is a plethora of layout methods, we selected these four layout methods because they are capable of producing diverse layouts by using the wide range of parameter values. Also, their publicly available, robust implementations did not produce any degenerate case in our evaluations. Lastly, these methods are efficient for computing a large number of layouts in a reasonable amount of time.

The resulting 20,000 layouts vary in many ways ranging from aesthetically pleasing looks to incomprehensible ones (e.g., hairballs). We did not remove any layout as we wanted to observe how the models encapsulate the essence of graph layouts from the diverse layouts.

3.3.2 Models and Configurations

We compare eight different model designs to investigate the effects of graph neural networks (GNNs) and Gromov–Wasserstein (GW) [160] distance in our model. All the models we use have the same architecture as described in Section 3.2.4, except for the GNN layers.

MLP: This model uses 1-layer perceptrons instead of GNNs; it does not consider the structure of the given graph. The model serves as a baseline for the investigation of the representational power of GNNs.

GCN: This model uses graph convolutional networks (GCN) [135] as the GNN layers. GCN is one of the early works of GNN.

GIN-1: This model uses graph isomorphism networks (GIN) [234] as the GNN layers. 1-layer perceptrons are used as the MLP in Equation 3.2.

GIN-MLP: This model uses GIN [234] as the GNN layers and 2-layer perceptrons as the MLP in Equation 3.2. We use the element-wise mean for aggregating the representation of neighbors of a node in both the GIN-1 and GIN-MLP models (f in Equation 3.2).

Also, for the models that use the GW distance in the optimization process, we add '+GW' to its model name. For example, GIN-MLP+GW denotes a model uses GIN-MLP as the GNN layers and the GW distance for comparing the structurally equivalent nodes

(SENs) in the optimization process. The models with GW [160] are only used for the graphs having SENs (all the graphs except can96 and football).

We use the L_1 loss function for the reconstruction loss. For the variational loss, we draw the same number of samples as the batch size from the prior distribution and set $c(x, y) = \|x - y\|_2^2$ to compute the sliced-Wasserstein distances (Equation 3.3). Also, we set $\beta = 10$ in the optimization objective (Equation 3.4).

We use varying numbers of hidden units in GNN layers depending on the number of nodes: 32 units for lesmis, can96, football and rajat11, 64 units for jazz, netsci, and dwt419, and 128 units for asoiaf and bus1138. The batch size also varies: 40 layouts per batch for asoiaf and bus1138, and 100 for the other graphs.

All the models use three GNN layers (or perceptrons in the MLP models), the exponential linear unit (ELU) [52] as the non-linearity, batch normalization [122] on every hidden layers, and an element-wise mean pooling as the readout function in the encoder. We use the Adam optimizer [190] with a learning rate of 0.001 for all the models. We train each model for 50 epochs.

3.3.3 Implementation

We implemented our models in PyTorch [174]. The machine we used to generate the training data and to conduct the evaluations has an Intel i7-5960X (8 cores at 3.0 GHz) CPU and an NVIDIA Titan X (Maxwell) GPU. The implementation of each layout method used in the evaluations is also shown in Table 3.2.

3.3.4 Test Set Reconstruction Loss

We compare the test layout reconstruction to evaluate the models' generalization capability. Here, a layout reconstruction means the model takes the input layout, encodes it to the latent space, and then reconstructs it from the latent representation. The test reconstruction loss quantifies the generalization ability of a model because it measures the accuracy of reconstructing the layouts that the model did not see in the training. We perform 5-fold cross-validations to compare the eight model designs in terms of their test set reconstruction loss. To reduce the effects of the fold assignments, we repeat the

Method	Parameter Ranges	Implementation
D3 [34]	link distance $= [1.0,\ 100.0]$, charge strength $= [-100.0,\ -1.0]$, velocity decay $= [0.1,\ 0.7]$	[33]
FA2 [123]	gravity $= [1.0,\ 10.0]$, scaling ratio $= [1.0,\ 10.0]$, adjust sizes $= \{\text{true},\ \text{false}\}$, linlog $= \{\text{true},\ \text{false}\}$, outbound attraction distribution $= \{\text{true},\ \text{false}\}$, strong gravity $= \{\text{true},\ \text{false}\}$	[21]
FM^3 [98]	force model $= \{\text{new},\ \text{fr}\}$, galaxy choice $= \{\text{lower mass},\ \text{higher mass},\ \text{uniform}\}$, spring strength $= [0.1,\ 1000.0]$, repulsive spring strength $= [0.1,\ 1000.0]$, post spring strength $= [0.01,\ 10.0]$, post repulsive spring strength $= [0.01,\ 10.0]$	[49]
sfdp [116]	repulsive force strength (C) $= (0.0, 5.0]$, repulsive force exponent (p) $= [0.0, 5.0]$, attractive force strength (mu) $= (0.0, 5.0]$, attractive force exponent (mu_p) $= [0.0, 5.0]$	[176]

Table 3.2. The layout methods and parameter ranges used for producing training data in our experiments. The parameter names follow the documentation of the implementations.

experiment 10 times and report the mean losses.

Results We compare the eight models in terms of the average reconstruction loss (lower is better) of the test sets, which are the 4,000 layouts that are not used to train the models. The results shown in Figure 3.4 are the mean test reconstruction losses of the 10 trials of the 5-fold cross-validations. The standard deviations are not shown as the values are negligible: all standard deviations are less than 3×10^{-4}.

Overall, the models with GIN-MLP and GW show the lowest loss of all the graphs. GIN-MLP models show the lowest loss for the graphs that do not have any structurally equivalent nodes, as shown in Figure 3.4a, where the models with GW are not used. Thus, GIN-MLP+GW models show the lowest loss for all the other graphs (Figure 3.4b–d).

Although the absolute differences vary, the ranking between the models based on

Figure 3.4. Average test reconstruction losses. Since the value ranges vary for each graph, we use individual ranges for each graph. The models with Gromov–Wasserstein distance [160] are not used for can96 and football as they do not have any structurally equivalent nodes.

the neural network modules is consistent in all the graphs. The ranking of the models with GW is as follows: GIN-MLP+GW, GIN-1+GW, GCN+GW, and MLP+GW. The ranking of the models without GW is as follows: GIN-MLP, GIN-1, GCN, and MLP. In addition, the three different rankings of the models are found for the graphs with structural equivalences (Figure 3.4b–d).

Discussion In lesmis, netsci, and dwt419 (Figure 3.4b), all the models with GW show a lower loss than the models without GW. We have found that many nodes in the three graphs have structural equivalences ($|S|/|V|$ in Table 3.1). This shows that the GW helps the learning process, especially for the graphs that have many structural equivalences.

For jazz and asoiaf (Figure 3.4c), the models with GIN show better results than the models without GIN. Although they also have a considerable ratio of structural equivalences, GIN-MLP and GIN-1 show lower reconstruction losses than GCN+GW and MLP+GW. Considering the number of nodes as shown in Table 3.1, we have found that the two graphs (jazz and asoiaf) are dense and have a relatively small average path length compared to the other graphs. This suggests the GIN is important for dense and

"small-world"-like networks [227].

For rajat11 and bus1138 (Figure 3.4d), the ranking of the models is as follows: GIN-MLP+GW, GIN-MLP, GIN-1+GW, GIN-1, GCN+GW, GCN, MLP+GW, and MLP. We have found that these two graphs are sparse and they have a small ratio of nodes with structural equivalence. This suggests that the representational power of the GNNs has a stronger effect than the usage of GW in such graphs.

3.3.5 Layout Metrics and Test Reconstruction Loss

To investigate our models' behavior on reconstructing the test sets, we analyze the correlations between the test reconstruction loss and each of the two layout quality metrics of the test input layouts: crosslessness [183] and shape-based metric [72]. Crosslessness [183] is a normalized form of the number of edge crossings where a higher value means fewer edge crossings. Shape-based metric [72] measures the quality of a layout based on the similarity between the graph and its shape graph (a relative neighborhood graph of the node positions). We use the Gabriel graph [84] as the shape graph of a layout. We use the models with the lowest test reconstruction loss for each graph from Section 3.3.4.

Results The Pearson correlation coefficients show that the test loss has strong negative correlations with both the crosslessness (c) and the shape-based metric (s) (i.e., when the reconstruction loss of a test input layout is low then its layout metrics are high). All the results are statistically significant as all the p-values are less than 2.2×10^{-16}:

	lesmis	can96	football	rajat11	jazz	netsci	dwt419	asoiaf	bus1138
c.	$-.69355$	$-.704$	$-.703$	$-.521$	$-.877$	$-.488$	$-.586$	$-.782$	$-.517$
s	$-.69364$	$-.812$	$-.350$	$-.707$	$-.749$	$-.630$	$-.715$	$-.311$	$-.706$

Discussion The results show that our models learn better on how to reconstruct the layouts with fewer edge crossings and are similar to its shape graph. In other words, our models tend to generate "good" layouts in terms of the two metrics, even if the models are not trained to do so. We show detailed examples in Section 3.3.6.

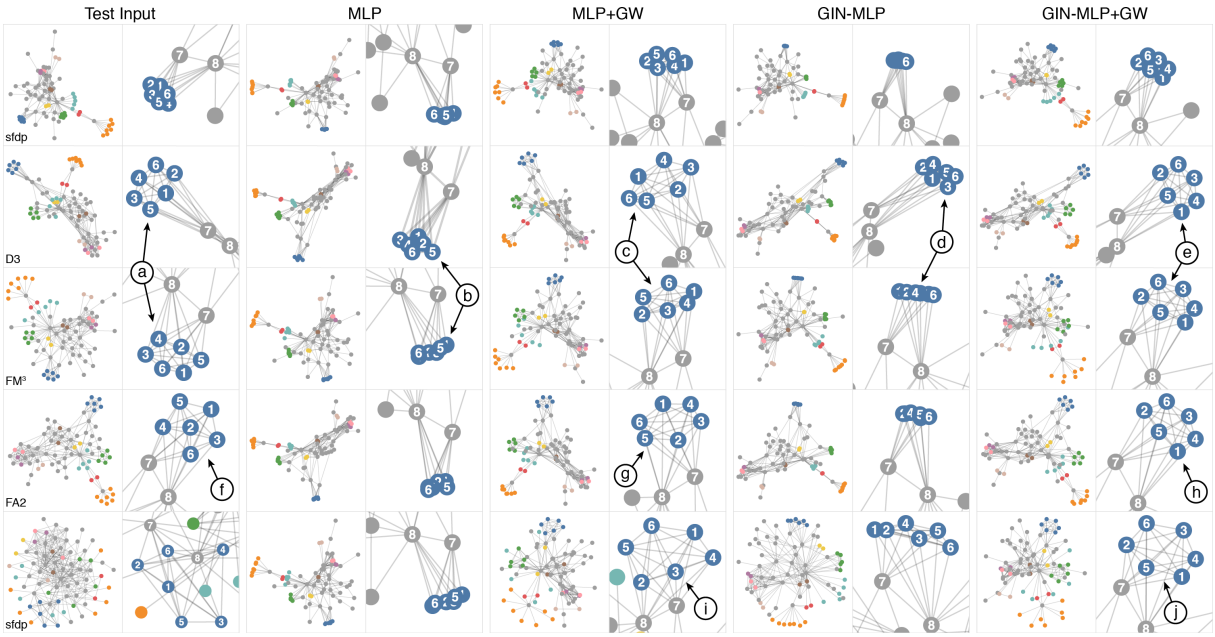


Figure 3.5. Qualitative results of the lesmis graph using the four different models. The leftmost column shows the test input layouts that the models did not see in their training session. The test input layouts are computed with randomly assigned parameter values as described in Section 3.3.1. The other columns show the reconstructed layouts of the test inputs using the four models. The nodes with the same color, except gray, are structurally equivalent to each other. The nodes in gray are structurally unique. The results are discussed in detail in Section 3.3.6.

3.3.6 Qualitative Results

We show the qualitative results of the layout reconstruction and the learned latent space to discuss the behaviors of the models in detail.

GIN-MLP and GW The models with GIN-MLP and GW show the lowest test reconstruction loss in Section 3.3.4. To further investigate, we compare the reconstructed layouts of the lesmis graph, which has the widest range of losses among the different models. Due to space constraints, we discuss the difference between the four different models (MLP, MLP+GW, GIN-MLP, and GIN-MLP+GW) on the lesmis graph.

Figure 3.5 shows the qualitative results of reconstructing unseen test input layouts, where a good generative model can produce diverse layouts similar to the unseen input layouts. The lesmis graph has a number of sets of SENs. In Figure 3.5, the nodes with the same color, except gray, are structurally equivalent to each other; they have the same

set of neighbors in the graph. The nodes in gray are structurally unique; each of them has a unique relationship to the other nodes in the graph. For example, the blue nodes (1–6) have the same relationship where they are all connected to each other and to the two other nodes (7 and 8).

As described in Section 3.2.3, the locations of SENs are often not consistent in different layouts. For example, the arrangements of the blue nodes in the D3 layout and the FM^3 layout are different (Figure 3.5a). In the D3 layout, the blue nodes are placed in the following clockwise order: 2, 1, 5, 3, 4, and 6, where the nodes $\{2, 1, 5\}$ are closer to $\{7, 8\}$ than $\{3, 4, 6\}$. However, in the FM^3 layout, the clockwise order is 4, 2, 5, 1, 6, and 3, where the nodes $\{4, 2, 5\}$ are closer to $\{7, 8\}$ than $\{1, 6, 3\}$.

The arrangements of the SENs in the reconstructed layouts vary depending on the model. For example, the models with GW are able to lay out the blue nodes (Figure 3.5c and e) similar to the input layouts (Figure 3.5a). However, the models without GW fail to learn this and produce collapsed placements (Figure 3.5b and d). We suspect this is due to the many possible permutations between a set of SENs. The models without GW tend to place a set of SENs at the average position of the SENs to reduce the average loss. However, the models with GW learn about the generalized concept of the blue nodes' arrangements and produce similar arrangements as the test input layouts in a different permutation of the SENs. The other sets of SENs show similar results (e.g., the orange and green nodes in Figure 3.5), where the models without GW produce collapsed arrangements, but the models with GW produce similar arrangements as the test input layouts.

In addition, the placements of the blue nodes are spatially consistent across the different layouts using GIN-MLP+GW (Figure 3.5e, h, and j) than using MLP+GW (Figure 3.5c, g, and i). This shows that the models with GIN-MLP gain a more stable generalization of the arrangement of SENs than the models with only MLP. Therefore, using GIN-MLP+GW models, users can generate diverse layouts while preserving their mental map across different layouts. This is not possible in many existing layout methods due to their nondeterministic results [169].

There are other examples of the generalization capability of our models. For example, Figure 3.5f shows a different arrangement of the blue nodes from Figure 3.5a, where the layout in Figure 3.5f has a star-like arrangement with node 2 in the center. However, the reconstructed layouts (Figure 3.5h) of the layout in Figure 3.5f are more similar to the layouts in Figure 3.5a. We have found that arrangements similar to the layouts in Figure 3.5a are more dominant in the training set.

Another example is the last row of the Figure 3.5, where the input layouts are hairballs. In contrast, the reconstructed layouts using our models are more organized. This also shows that the models have a generalized concept of graph layouts. This might explain the negative correlations in Section 3.3.5 as the models do not reconstruct the same hairball layouts.

Multiple Graphs To demonstrate our models with more graphs, Figure 3.7 shows the qualitative results of five different graphs using the GIN-MLP model for the football graph and the GIN-MLP+GW models for the other graphs. The first three rows show that our models are capable of learning different styles of layouts. The bottom two rows show the models' behavior on hairball-like input layouts. We can see that the reconstructed layouts are more organized. For example, the fourth layout of dwt419 is twisted. However, the reconstructed layout is not twisted but pinched near the center, like other layouts. These examples might explain the negative correlations in Section 3.3.5 as the models do not reconstruct hairball layouts well in favor of generalization.

Latent Space A common way to qualitatively evaluate a generative model is to show the interpolations between the different latent variables [92, 97, 134, 139, 193]. If the transitions between the generated samples based on the interpolation in the latent space are smooth, we can conclude that the generative model has a generalization capability of producing new samples that were not seen in training.

As our models are trained to construct a two-dimensional latent space, we can show a grid of generated samples interpolating throughout the latent space. We show the results of can96 and rajat11 in Figure 3.8 and the result of the lesmis in Figure 3.1. As

Name	D3	FA2	FM^3	sfdp	Epoch
lesmis	.195	.620	.015	.229	76.7
can96	.239	.695	.021	.329	34.7
football	.302	.845	.033	.486	43.9
rajat11	.358	.856	.044	.586	45.1
jazz	.599	1.61	.116	1.18	212
netsci	1.14	2.50	.236	2.59	281
dwt419	1.24	3.08	.257	3.08	211
asoiaf	3.03	9.42	.620	9.16	690
bus1138	4.41	7.88	1.07	10.3	509

Table 3.3. Computation time. The table shows the layout computation times (D3, FA2, FM^3, and sfdp) for collecting the training data and training the model (Epoch). The layout computation times are the mean seconds for computing one layout per method per graph. The training computation times are the mean seconds for training one epoch (16K samples) per graph.

we can see, the transitions between the generated samples are smooth. Also, the latent space of can96 is particularly interesting. It seems that the model learned to generate different rotations of the 3D mesh. However, all of the layouts used in this chapter are 2D. In addition, the input feature of a layout is the normalized pairwise distances between the nodes, as described in Section 3.2.2, which do not explicitly convey any notion of 3D rotation.

Based on these findings, we conclude that our models are capable of learning the abstract concepts of graph layouts and generating diverse layouts. An interactive demo is available in the supplementary material [10] for readers to explore the latent spaces of all the graphs using all the models we have described in this chapter.

3.3.7 Computation Time

We report and discuss the layout computation time, model training time, and layout generation time of each graph using the models with the lowest test reconstruction loss in Section 3.3.4. The layout computation time and model training time are shown in Table 3.3.

As we collect a large number of layouts (16K samples per graph for training), computing layouts is the most time-consuming step in building the models. However,

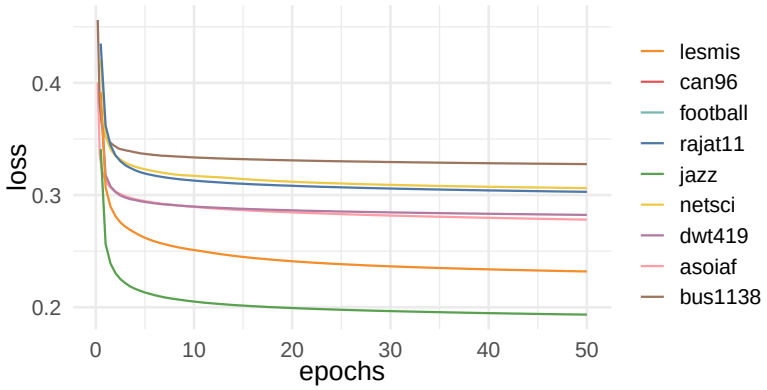


Figure 3.6. The average training loss per batch shows that our models converge quickly.

we can incrementally generate training examples and train the model simultaneously.

We have found that the number of nodes having structural equivalence ($|S|$ in Table 3.1) is a strong factor to the training time. It is directly related to the complexity of the GW distance [160] computation (Section 3.2.3). If we can build a generative model using GANs [92], instead VAEs [134] (SWAE [139] in this chapter), we can remove the GW distance computation in the process (more in Section 3.5). This can significantly reduce the training time. Also, the number of edges might be a stronger factor to the training time than the number of nodes as we have implemented GNNs using sparse matrices. For a small graph (e.g., $|V| < 200$), it might be faster to use dense matrices for GNNs.

In addition, the models converge quickly as shown in Figure 3.6. Although the models are trained for 50 epochs for our evaluations, the models are capable of generating diverse layouts less than 10 epochs.

After training, the mean layout generation times for all the graphs are less than .003 s. Thus, users can explore and generate diverse layouts in real time, which is demonstrated in the supplementary material [10].

Based on these findings, we expect training a new model for a graph from scratch without any layout examples can be done within a few minutes for a small graph ($|V| < 200$, $|E| < 500$) and a few hours for larger graphs. Although it is a considerable computation time, our model can be trained in a fully unsupervised manner; it does not require users to be present during the training. Thus, our approach can save the user's

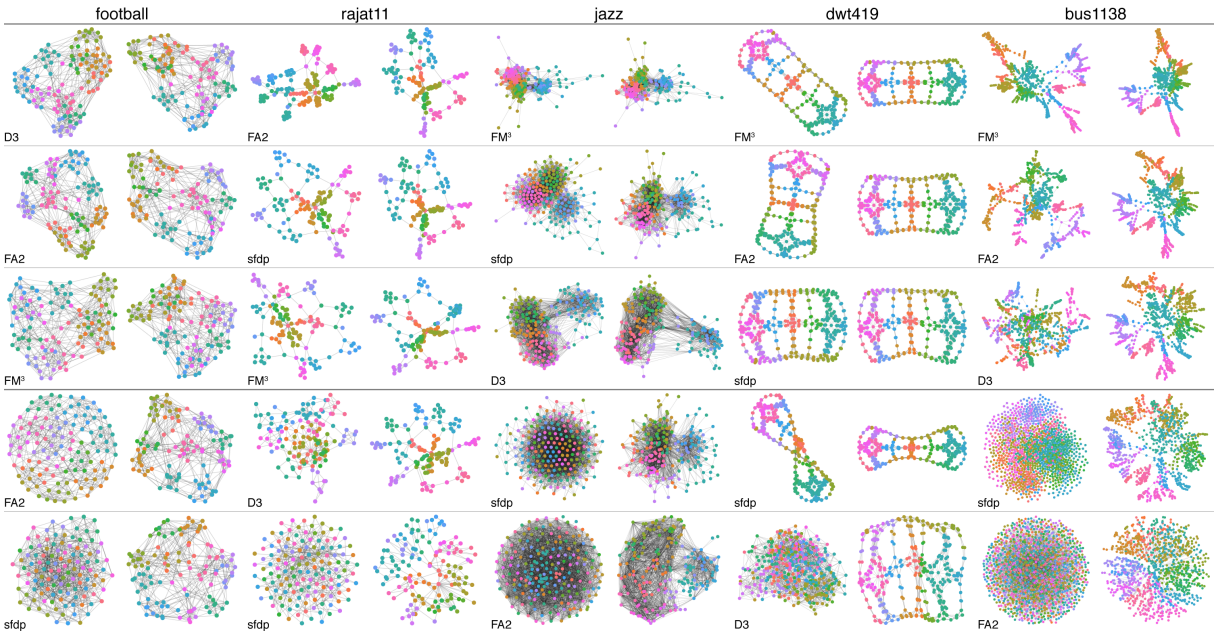


Figure 3.7. Qualitative results of the five different graphs. The GIN-MLP model is used for the football graph, and the GIN-MLP+GW models are used for the other graphs. For each pair of layouts, the left is the test input, and the right is the reconstructed layout. The first three rows show the different styles of layouts for each graph, and the bottom two rows show the reconstruction results of hairball layouts. The results are discussed in Section 3.3.6.

time—which is much more valuable than the computer's time—by preventing them from blindly searching for a good layout.

3.4 Usage Scenario

The previous section shows that our model is capable of generating diverse layouts by learning from existing layouts. This section describes how users can use the trained model to produce a layout that they want.

After training, users can generate diverse layouts of the given graph by feeding different values of the latent variable (z_L in Section 3.2.4) to the decoder. Thus, the interpretability of the latent space is important for users to easily produce a layout that they want. We achieve this by using a 2D latent space rather than a higher-dimensional space. A 2D latent space is straightforward to map additional information onto it.

By mapping generated samples on the 2D latent space (e.g., the sample grid in

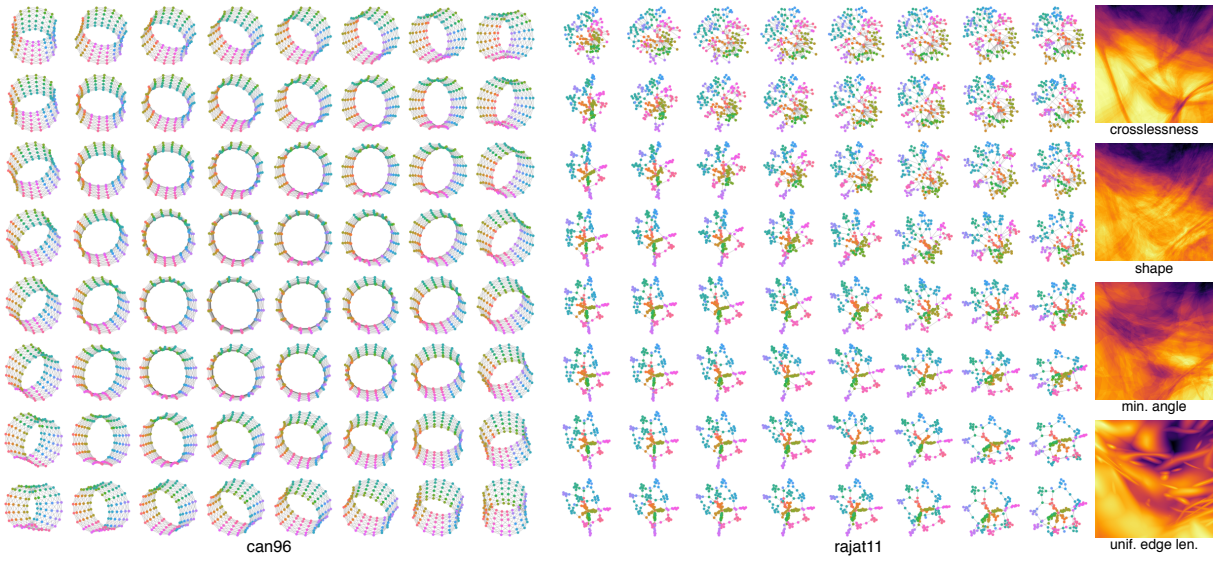


Figure 3.8. Visualization of the latent spaces of can96 and rajat11. The GIN-MLP model is used for can96 and the GIN-MLP+GW model is used for rajat11. The grids of layouts are the generated samples by the decoding of a 8×8 grid in $[-1, 1]^2$. Also, Figure 3.1 shows the sample grid of lesmis. The smooth transitions between the generated layouts show that the capability of generalization of our models. Novices can directly use this as a WYSIWYG interface to generate a layout they want. The rightmost column shows the heatmaps of the four layout metrics of 540×540 generated sample layouts in $[-1, 1]^2$ of rajat11. Experts can use the heatmaps to see complex patterns of layout metrics on diverse layouts. The results are discussed in detail in Section 3.3.6 and Section 3.4. The results of other graphs and models are available in the supplementary material [10].

Figure 3.1 and Figure 3.8), we can build a what-you-see-is-what-you-get (WYSIWYG) interface for users to intuitively produce a layout that they want. Finding a desired layout from an unorganized list of multiple layouts (e.g., the training samples in Figure 3.1) often results in a haphazard and tedious exploration [29]. However, the sample grid provides an organized overview with a number of representative layouts of the given graph. With the sample grid as a guide, users can intuitively set the latent variable (z_L) to produce a suitable layout by pointing a location in the 2D latent space, as shown in Figure 3.9. They also can directly select the desired one from the samples. The WYSIWYG interfaces of each graph are demonstrated in the supplementary material [10].

Moreover, our approach produces spatially stable layouts. As we have discussed in

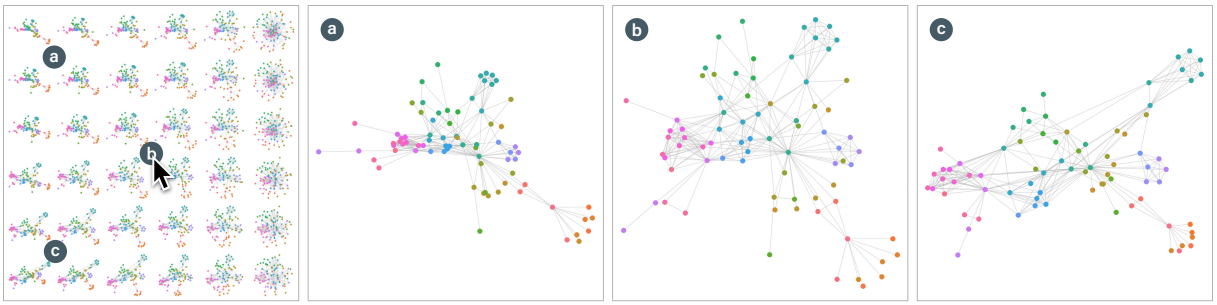


Figure 3.9. Users can intuitively produce the layout they want by simply pointing to a location in the latent space (left most), without blindly tweaking parameters of layout algorithms. An interactive demo is available in the supplementary material [10].

Section 3.2.2, many layout methods are nondeterministic. For example, in the training samples of Figure 3.1, the locations of the orange nodes vary greatly across different layouts. Thus, identifying the same node(s) among these layouts is difficult because region-based identifications cannot be utilized [169]. Comparing nondeterministic layouts often requires a considerable amount of the user's time since they need to match the nodes between different layouts. However, as shown in Figure 3.1 and Figure 3.8, our models produce spatially stable layouts, where the same node is placed in similar locations across different layouts. Hence, identifying the same node(s) in different layouts is straightforward, and thus comparing layouts becomes an easy task.

Using our approach, users can directly see what the layout results will look like with the sample grid. Also, spatially-stable layout generation enables users to effortlessly compare various layouts of the given graph. Thus, users can intuitively produce a layout that best suits their requirements (e.g., highlighting the interconnections between different communities) without blindly tweaking parameters of layout methods.

By mapping layout metrics on the latent space, users can directly see the complex patterns of the metrics on diverse layouts of a graph. Figure 3.8 (the rightmost column) shows heatmaps of four layout metrics of 540 $\times$ 540 layouts of rajat11. While the sample grid shows smooth transitions between different layouts, the heatmaps show interesting patterns. For example, there are several steep "valleys" in the heatmap of crosslessness [183], where the darker colors mean more edge crossings. This shows crosslessness is sensitive to certain changes in the layouts. Using the heatmap as

an interface, users can exactly see these changes through producing several layouts by pointing the locations across the valleys in the heatmap. Thus, experts in graph visualization can use the heatmaps for designing layout metrics as they can understand how layout metrics behave on various layouts with concrete examples.

Our approach is an example of *artificial intelligence augmentation* [44], where our generative model builds a new type of user interface with the latent space to augment human intelligence tasks, such as creating a good layout and analyzing the patterns of layout metrics.

3.5 Discussion

Section 3.3 and Section 3.4 have discussed the evaluation results and usage scenarios. This section discusses the limitations and future research directions of our approach. This chpater has introduced the first approach to generative modeling for graph layout. As the first approach in a new area, there are several limitations we hope to solve in the future.

The maximum size of a graph in our approach is currently limited by the capacity of GPU memory. This is the reason we could not use a larger batch size for asoiaf and bus1138 (40 layouts per batch, while 100 layouts per batch for the other graphs). As a sampling-based GNN [59] scalable to millions of nodes has been recently introduced, we expect our approach can be applied to larger graphs in the future.

Although our model is capable of generalizing for different layouts of the same graph, it does not generalize for both different graphs and different layouts. Therefore, we need to train a new model for each graph. Our model can be trained in a fully unsupervised manner and can be trained incrementally while generating the training samples simultaneously. A better model would learn to generalize across different graphs so that it can be used for any unseen graphs. However, this is a very challenging goal. Most machine learning tasks that require the generalization across different graphs (e.g., graph classifications) aims to learn graph-level representations. But the generalization across both graphs and their layouts in a single model requires to learn

the latent representations of nodes across many different graphs. Unfortunately, this is still an open problem.

Our model learns the data distribution of the training set. However, a valid layout can be ignored in favor of generalization. For example, the arrangement of the blue nodes in Figure 3.5f is a valid layout, but it is a rare type of arrangement in the training dataset. Thus, the model produces a more general arrangement following the distribution of the training set (Figure 3.5h). As a valid layout can be an outlier in the training dataset, we need an additional measure to not over-generalize valid layouts in the training dataset and properly reconstruct valid layouts.

We have used a sliced-Wasserstein autoencoder [139], a variant of VAE [134], for designing our architecture. As a VAE, we explicitly define a reconstruction loss for the training. However, this was challenging for comparing two different layouts of a set of structurally equivalent nodes. In this chapter, we have used the GW distance [160] to address this issue. Another possible solution is to use GANs, which do not require an explicit reconstruction loss function in the optimization process. Thus, using a GAN can reduce the computational cost because computing GW distance is no longer required. While we did not use GANs due to mode collapse and non-convergence [91], we believe it is possible to use GANs for graph layout in the future.

In this chapter, we map a number of generated samples on a 2D latent space to directly see the latent space. However, the learned latent representation of our model is *entangled*, which means each dimension of the latent space is not interpretable. As we can see from the grids of samples in Figure 3.1 and Figure 3.8, although we can see what the generated layouts look like with different latent variables, we cannot interpret the meaning of each dimension. Thus, it is difficult to use a higher-dimensional latent space for our purpose, as we cannot either interpret each dimension or see the overview of the latent space. Learning a model that produces disentangled representations is an important research direction in generative modeling [111]. With a generative model that can learn a disentangled latent space, we can produce a layout in a more interpretable way, where each dimension only changes a specific aspect of a layout independently.

For example, if one dimension of the latent representation encodes the area of clusters of nodes, we can directly manipulate a layout to change the cluster size of a layout.

3.6 Conclusion

Graph-structured data is one of the primary classes of information. Creating a good layout of a graph for visualization is non-trivial. The large number of available layout methods and each method's associated parameter space confuse even the experts. The trial-and-error efforts require a significant amount of the user's time. We have introduced a fundamentally new approach to graph visualization, where we train a generative model that learns how to visualize a graph from a collection of examples. Users can use the trained model as a WYSIWYG interface to effortlessly generate a desired layout of the given graph.

Generative modeling for image datasets has shown dramatic performance improvement; it took only four years from the first model of generative adversarial networks [92] to an advanced model that can generate high-resolution images [128]. There can be many exciting ways to use generative models for graph visualization, or even other types of data visualization. We hope this chapter will encourage others to join this exciting area of study to accelerate designing generative models for revolutionizing visualization technology.

Chapter 4

Augmenting the Design Process of Adjacency Matrix Views using a Deep Generative Model

A node-link diagram represents a node as a point and an edge as a line segment connecting the two points associated with its incident nodes. Another popular way to visualize a graph is visually encoding an adjacency matrix. An adjacency matrix A is a square matrix, where the rows and the columns correspond to the nodes, and an element $A_{i,j}$ indicates the edge between node u_i and node u_j. For a simple, undirected, unweighted graph, $A_{i,j}$ (and $A_{j,i}$) is 1 if node u_i and node u_j are connected; otherwise, $A_{i,j} = A_{j,i} = 0$.

An adjacency matrix view is a visual representation of an adjacency matrix by coloring an area mark for each element of the matrix based on its value [169]. Although node-link diagrams are more popularly used for visualizing graphs, adjacency matrix views are often preferred for dense graphs. Matrix views do not suffer from occlusion issues of node-link diagrams, such as node overlaps and edge crossings. Thus, adjacency matrix views can have higher perceptual scalability and information densities than node-link diagrams.

Different node-link diagrams can show distinct structural characteristics of the same graph depending on how the nodes are laid out in a two- or three-dimensional space.

In an adjacency matrix view, nodes are laid out linearly in terms of their order along the rows (and the columns) of the matrix; and different node orderings often highlight different structural information of the same graph (e.g., Figure 4.1). Therefore, it is essential to find those "good" node orderings that show the aspects of a graph that users wish to spotlight.

Decades of research have introduced many matrix reordering methods, which sort the rows and the columns of a matrix to reveal the structural information in the data [22, 151, 231]. However, there is no "best" method, as each method follows a different set of heuristics that can reveal certain structures in a graph. Therefore, users typically try multiple reorderings of the given adjacency matrix using different methods until they find one that best fits a specific visualization goal. Unfortunately, this trial-and-error approach is time-consuming and haphazard, which is especially challenging for novices.

This chapter presents a technique that builds an intuitive interface for users to effortlessly find the desired adjacency matrix view of a graph. The fundamental approach follows the one for node-link diagrams presented in Chapter 3. We first collect a set of reordering examples of the adjacency matrix of the given graph. Then, we train a generative model that learns a latent space of diverse matrix reorderings of the given graph. Finally, based on a grid of samples from the learned space, we build a what-you-see-is-what-you-get (WYSIWYG) interface that enables users to intuitively depict diverse reorderings of the given adjacency matrix.

Although the idea of building a WYSIWYG interface using the learned latent space of a deep generative model is the same as Chapter 3, there are several unique challenges in designing a neural network architecture that can reorder an adjacency matrix. As reordering a matrix is not a differentiable problem, we need a differentiable approximation to train a neural network with backpropagation. However, a naive approximation of an adjacency matrix can be invalid, which does not represent the same structure as the given graph. In addition, depending on the automorphisms of the given graph, some distinct node orderings may produce the same result.

This chapter designs an encoder-decoder architecture and its training scheme that addresses these challenges. Some key design elements are as follow:

- During training, the model generates "soft" permutation matrices using a relaxed sorting operator to enable gradient-based learning.
- To ensure the resulting matrices are valid adjacency matrices of the given graph, the decoder learns to produce permutation matrices and then permutes the given adjacency matrix.
- The loss function takes only the resulting reordered matrices into account and does not consider how the nodes are permuted.

Our evaluation results show that the presented architecture is capable to learn a generative model that can visualize a graph in diverse matrix reorderings.

Most existing research in this area generally focused on developing algorithms that can compute "better" matrix reorderings for particular circumstances. This chapter introduces a fundamentally new approach to matrix visualization of a graph, where a machine learning model learns to generate diverse matrix reorderings of a graph. A grid of samples from the learned latent space shows representative matrix reorderings of the given graph. Therefore, users can effortlessly find the desired matrix reordering using the presented technique to design effective matrix visualization of a graph for meeting a given analysis goal.

4.1 Related Work

Designing a deep generative model for matrix reordering is related to matrix reordering methods, deep generative modeling, and deep learning for sorting. This section focuses on matrix reordering methods and deep learning techniques for sorting as Section 3.1.2 already discussed deep generative modeling.

4.1.1 Matrix Reordering

Matrix reordering is a problem of finding a "good" ordering of the rows and columns of a relational matrix (e.g., an adjacency matrix and a correlation matrix) to reveal structural information from a set of objects. This problem is also known as (or equivalent to)

seriation, *linear arrangement*, and *sequencing*. Since there can be $n!$ different permutations of a set of n objects, a brute-force enumeration approach to finding a good node ordering is infeasible even for a graph with a few dozens of nodes.

Decades of research across various disciplines have introduced many methods for matrix reordering, where each method uses a different set of heuristics, such as traveling salesman problem solvers [230], spectral ordering (i.e., the order of the Fiedler vector) [20, 68], and hierarchical clustering [16, 75, 96]. Several studies [22, 101, 151, 231] have reviewed various matrix reordering methods and have shown that different methods can show distinct patterns in the same matrix.

Unfortunately, it is not clear how to derive a suitable reordering for the given dataset and analysis goal [23, 24]. Users typically first produce multiple reorderings of the given matrix using various methods and then select one that fits the given circumstances. Several techniques have been introduced to support users through such a trial-and-error process by providing interactive methods to steer matrix reordering algorithms [24, 43, 109, 212]. The goal of these techniques is to narrow down the search space. Therefore, multiple trials with continuous human intervention would be required to explore other possible reorderings of the same matrix. In contrast, our approach is to build a model that can produce diverse reorderings of the given matrix, not to narrow the search space. Moreover, we train such a model in a fully unsupervised manner, thus not requiring human intervention.

4.1.2 Deep Learning for Sorting

Sorting is one of the most important problems in computer science. Matrix reordering is also a sorting problem, where the goal is to find an ordering that can uncover structural information in a relational matrix. For our goal, we need to train a neural network that can produce diverse reorderings of the adjacency matrix of the given graph. However, sorting is not differentiable with respect to the input. It is thus impossible to use end-to-end gradient-based optimization to train a neural network with exact sorting functions.

Recently, a number of relaxed (differentiable) sorting operators have been introduced

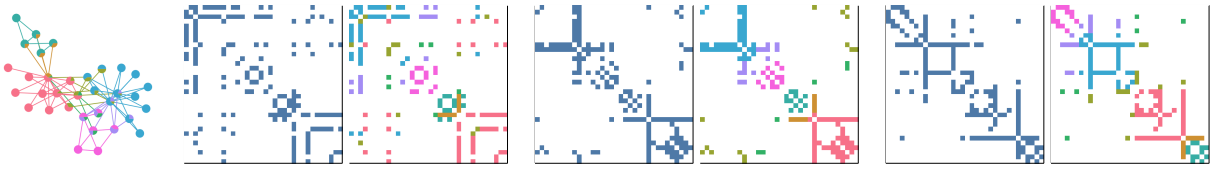
to enable gradient-based learning of models that include a sorting process [94, 161, 180]. The outputs of these relaxed sorting operators are "soft" permutation matrices. For example, the output of the Sinkhorn operator [161] is a doubly stochastic matrix, where the sum of any row or any column is 1. In addition, the NeuralSort [94] and SoftSort [180] operators produce a unimodal row stochastic matrix, where any row has the sum of 1 and has a unique argmax.

Using soft permutation matrices computed by the relaxed sorting operators, we can train a neural network model with approximated solutions. Once training is completed, these relaxed sorting operators can output "exact" permutation matrices to produce actual solutions for evaluation. For instance, since an output of the NeuralSort [94] and SoftSort [180] operators is a unimodal row stochastic matrix, we can simply apply row-wise argmax to get an exact permutation matrix. For a model with the Sinkhorn operator [161], an exact permutation can be obtained by passing the output of Sinkhorn operator (i.e., the output of the previous layer) to a linear assigment problem solver (e.g., Hungarian method [143]).

We use these relaxed sorting operators for learning a deep generative model that produces diverse reorderings of the adjacency matrix of a graph. In the evaluation, we compare two relaxed sorting operators for our problem, namely Sinkhorn [161] and SoftSort [180].

4.2 Approach

This chapter aims to design a technique for users to intuitively find the desired matrix view of a graph. More specifically, the goal is to learn a deep generative model that can systematically visualize a graph as multiple adjacency matrix views with diverse node orderings that can reveal different structural information of the same graph. This section describes the whole process of constructing a deep generative model for matrix reorderings, from gathering training samples to designing the neural network architecture.



HC-Average R2E SPIN-NH

Figure 4.1. Different structural patterns of the same graph can appear depending on the reordering methods. This example shows three different reorderings of the adjacency matrix of the karate graph (Table 4.1) using the shortest-path distances between the nodes. Throughout this chapter, the color of the multi-colored matrices and node-link diagrams represents the community structure of the graph [209], so the readers can easily compare different matrix reorderings, unless otherwise specified. However, this clustering-based color mapping is not commonly used. Thus, we also show single-colored matrices when possible, which better demonstrate how different orderings show distinct patterns in real-world matrix visualizations.

4.2.1 Training Data Collection

In order to learn a high-quality generative model that can produce various matrix reorderings, we need to collect a large and diverse set of matrix reorderings of the given graph. We collect such a dataset using various matrix reordering methods, multiple measures for computing distances between nodes, and random initial node orderings.

As we discussed in Section 4.1.1, decades of research have led to many matrix reordering methods. The adjacency matrix of the same graph can reveal different structural patterns depending on which method is used, as shown in Figure 4.1. Thus, an obvious way to collect diverse reorderings of the given graph is using multiple reordering methods.

This chapter focuses on a simple, undirected, unlabeled graph (see Section 2.1.1 for the definitions). Since an adjacency matrix of an undirected graph is a symmetric matrix, we can use a single node ordering for sorting both the rows and columns of the matrix. Figure 4.2 shows an overview of the matrix reordering process. To reorder the adjacency matrix A of an undirected graph, a matrix reordering method typically begin with a pairwise distance (or dissimilarity) matrix D, where $D_{i,j}$ represents the distance between nodes u_i and u_j, and $D_{i,i} = 0$. Therefore, depending on the measure of distance, we can get different reordering results with the same reordering method.

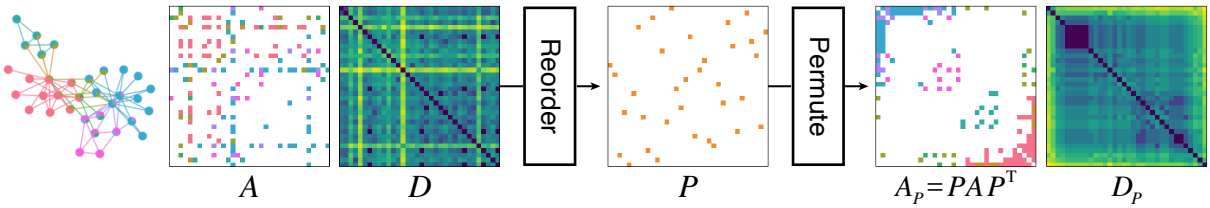


Figure 4.2. Reordering the adjacency matrix A of an undirected graph. A reordering method takes pairwise dissimilarities D between the nodes and computes a permutation P of the nodes minimizing a loss function to reveal structural information in the graph. In this example, two primary clusters (blue and red) are clearly visible in the reordered adjacency matrix A_P, but not in the unordered one A.

The shortest-path distance is one of the most common measures for calculating dissimilarity between nodes in a graph. Another common approach is considering the adjacency matrix of a graph as the feature vectors of nodes, where each row/column represents the corresponding node. Then, general distance (or dissimilarity) functions (e.g., Table 4.2) can be used to obtain the pairwise distance matrix, and the resulting matrix reorderings can vary greatly depending on the measure, as shown in Figure 4.3. We include the shortest-path distance and various distance functions (e.g., Table 4.2) to collect diverse reordering results of a graph.

When two nodes have the same set of adjacent nodes, their adjacency vectors (i.e., a row/column of the adjacency matrix) are the same. In this case, the distance between the two nodes can be calculated as 0, if the distance function only considers their adjacency vectors (e.g., Euclidean distance). In order to distinguish nodes with the same set of adjacent nodes, self-loops are often added (i.e., set the diagonal elements of the adjacency matrix to be 1) before computing the pairwise distance matrix, so

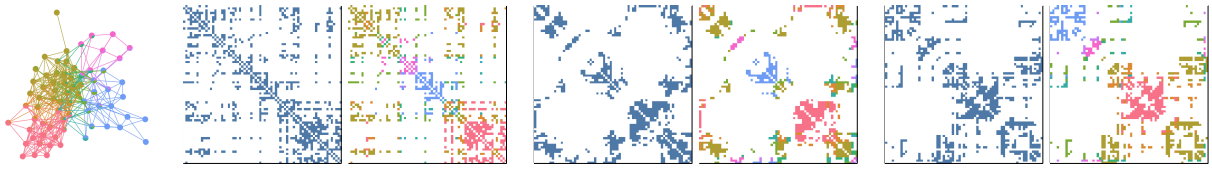
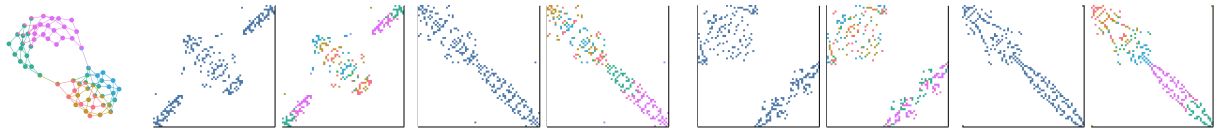


Figure 4.3. Different structural patterns of the same graph can appear depending on the distance measure, although the same reordering method is used. This example shows reordering results of the macaque graph (Table 4.1) using three different distance measures with the TSP [53, 230] method.



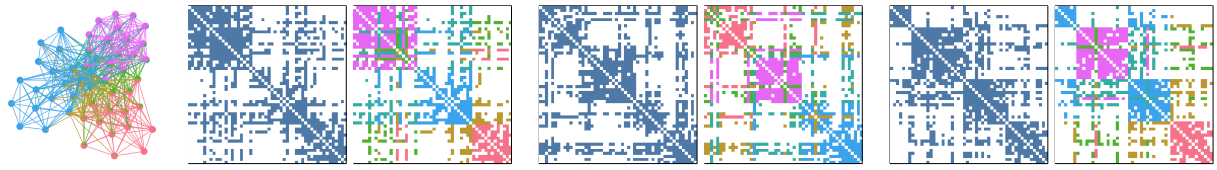
ARSA with A ARSA with $\hat{A}$ R2E with A R2E with $\hat{A}$

Figure 4.4. Different structural patterns of the same graph can appear by adding self-loops even with the same reordering algorithm and the same distance function. In this example, the matrix reorderings of the `gd96c` graph (Table 4.1) are computed with ARSA [38] and R2E [46] and the Jaccard distances between nodes. The results are quite different depending on whether the raw adjacency matrix (A) or the adjacency matrix with self-loops ($\hat{A}$) is used for computing distances between nodes. The adjacency matrix with self-loops ($\hat{A}$) is only used for computing distances between nodes, but not for visualizing a graph.

that the distance between two different nodes is greater than 0. The same reordering method with the same distance function can produce very different results depending on whether self-loops are added or not, as shown in Figure 4.4. We use both the (raw) adjacency matrix A and the adjacency matrix with self-loops $\hat{A}$ of the given graph when we compute the pairwise distances.

Lastly, several reordering methods often produce different results depending on the initial node orderings. Figure 4.5 shows such an example, where the same reordering method with the same distance measure produce different reordering results just because of the initial orderings of the given adjacency matrix are different. Thus, we also include a number of random initial orderings in when collecting a training dataset.

Obtaining a large collection of matrix reorderings can take a significant amount of time. However, similar to the data collection process of the previous chapter (Sec-



(a) (b) (c)

Figure 4.5. Different structural patterns of the same graph can appear depending on the initial node ordering even though the same reordering method and the distance measures betwen nodes are used (OLO-Average and shortest-path distance in this example). In the previous examples (Figure 4.1, Figure 4.3, Figure 4.4), the same initial node orderings were used for each graph.

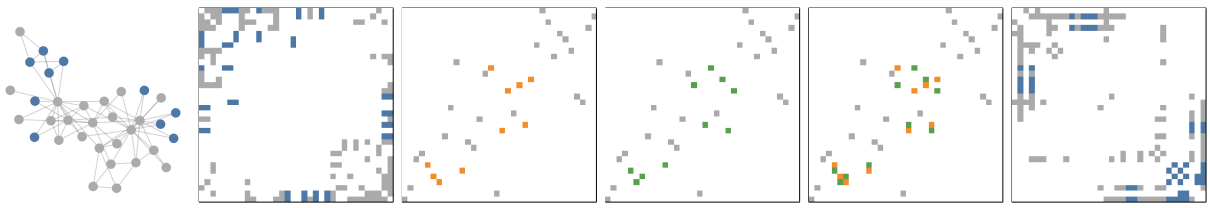
tion 3.2.1), the training process can begin as soon as we have a few dozen samples with mini-batch training schemes. Therefore, the model training process and the data collection process can be performed simultaneously and incrementally. With the incremental training process, users can utilize our generative model as quickly as possible.

4.2.2 Permutation Equivariance

A permutation of the nodes of a graph is said to be an automorphism if the permutation preserves the adjacency of each node. In other words, if the permutation P is an automorphism, the permuted adjacency matrix $A_P = PAP^\mathsf{T}$ is equal to the original adjacency matrix A ($0 = A_P - A$).

When a graph has a nontrivial automorphism (i.e., non-identity automorphisms), there are multiple different permutations that produce an equal reordered adjacency matrix. Figure 4.6 shows such an example, where two different permutations P_1 and P_2 of the karate graph produce the same reordered adjacency matrix ($0 = A_{P_1} - A_{P_2}$).

Hence, we need to design a neural network architecture and a loss function considering the fact that different permutations can produce an equal reordered adjacency matrix. However, finding all the permutations that produce the same reordered adjacency matrix is practically impossible since the graph automorphism problem belongs to the class NP of computational complexity, similar to the graph isomorphism problem. Therefore, the architecture and the loss function need to be permutation equivariant: they should not depend on how the nodes are permuted, but only on the resulting re-



(a) Layout (b) A (c) P_1 (d) P_2 (e) $P_1 \vee P_2$ (f) A_{P_1} and A_{P_2}

Figure 4.6. Permutation equivariance. Different permutations can produce an equal matrix reordering result depending on the structure of the given graph. In this example, although the permutations P_1 and P_2 are different, the reordered adjacency matrices A_{P_1} and A_{P_2} are equal. The orange and green cells are the difference in the two permutations P_1 and P_2 (c, d, and e), and the blue nodes and cells (a, b, and f) are the nodes in the graph associated with the difference.

ordered adjacency matrices. We thus train the model to reconstruct reordered adjacency matrices, not to reconstruct permutations (i.e., node orderings). We describe our design of the architecture and the loss function in more detail in Section 4.2.3.

4.2.3 Architecture

This section discusses an encoder-decoder architecture to learn a generative model that can produce diverse matrix reorderings of the given graph. The architecture follows the general framework of VAEs [134], which we discussed in the preliminaries in Section 3.2. Figure 4.7 shows an overview of the architecture.

Encoder The encoder's goal is to learn the low-dimensional latent representation z of a reordered adjacency matrix $A_P = \text{Reorder}(A, P) = PAP^\intercal$ (Figure 3.3a), where A is the original adjacency matrix and P is the permutation matrix that represents a node ordering. As we discussed in Section 4.2.2, we use a reordered adjacency matrix A_P as the input object rather than a permutation P because there can be different permutations that produce the same reordered adjacency matrix.

There is another intuition of using reordered adjacency matrices as the input objects: we can treat a reordered adjacency matrix as an image since it is a visual representation of the given graph designed to help the viewers comprehend the data. With this intuition, we design the encoder inspired by neural networks for learning a generative model of images [134, 186]. For example, Kingma and Welling [134] have designed a variational autoencoder using a multi-layer perceptron (a fully-connected neural network), where the input images are converted as one-dimensional vectors before they are fed to the encoder. In addition, Radford et al. [186] used a convolutional neural

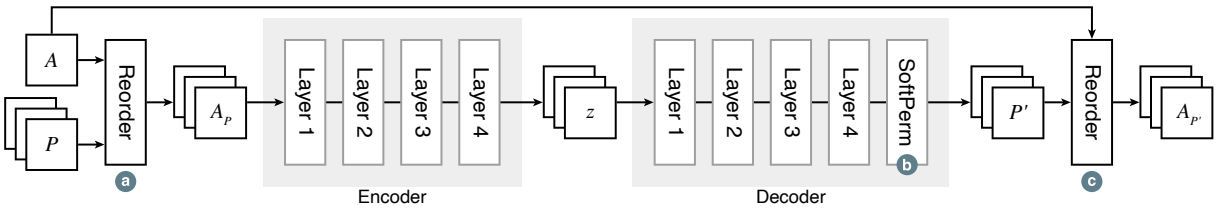


Figure 4.7. Our encoder-decoder architecture that learns a generative model from a collection of example matrix reorderings. We describe this architecture in Section 4.2.3

network, which has been widely used for various computer vision tasks [149], for designing a generative adversarial network [92].

Whether we use a multi-layer perceptron (MLP) or a convolutional neural network (Conv), the fundamental goal of the encoder is to learn a highly compressed representation z that captures the essence of a reordered adjacency matrix A_P, where each layer of the encoder gradually reduces the dimensionality of representations. In the evaluation (Section 4.3), we compare two encoder designs (MLP and Conv) for learning a generative model of matrix reordering.

We set the encoder to produce a two-dimensional representation of the input A_P. Similar to the latent space of node-link diagrams in the previous chapter (Figure 3.1), a two-dimensional latent space is straightforward to map a grid of generated samples on the latent space. Thus, the users can effortlessly interpret and navigate the latent space by using the grid of generated samples as a WYSIWYG interface. We use the uniform distribution of $[-1, 1]^2$ as the prior distribution so that the encoder learns to represent the input A_P in $[-1, 1]^2$.

Decoder The decoder's goal is to reconstruct the input objects from their latent representations—computed by the encoder. In general, the decoder is trained to reconstruct the input objects (e.g., images of handwritten digits [134]) directly from the trainable parameters (i.e., weights and biases) of the neural network. However, for an adjacency matrix, naively training the decoder to reconstruct an input (i.e., a reordered adjacency matrix A_P) from the decoder leads to producing an invalid adjacency matrix.

For general image data (e.g., landscapes, portraits, and animals), small errors in the generated results do not have a substantial effect on the viewers' understanding of the image content. In the case of an adjacency matrix view, however, even a small reconstruction error can lead to producing an invalid visualization that can mislead the viewers about the data, where the graph structure represented by a reconstructed adjacency matrix is different from the structure of the given graph. Therefore, we need to ensure that the decoder produces adjacency matrices representing exactly the same structure as the given graph.

We design the decoder to generate a permutation matrix P' and then reorder the given adjacency matrix A with the generated permutation matrix P'. Specifically, the input A_P is reconstructed as $A_{P'} = \text{Reorder}(A, P') = P'AP'^{\mathsf{T}}$, where P' is the output of the decoder and A is the original adjacency matrix (Figure 4.7c). In essence, the decoder learns to reconstruct the permutation matrix P (or its automorphism) of the input matrix reordering $A_P = PAP^{\mathsf{T}}$, not to directly reconstruct A_P.

For the decoder to produce a permutation matrix, we use a relaxed sorting operator that outputs permutation matrices, such as Sinkhorn [161] and SoftSort [180] (discussed in Section 4.1.2). A relaxed sorting operator is placed after the decoder's layers with trainable parameters (SoftPerm Figure 4.7b). We set the SoftPerm to produce "soft" permutation matrices during training to enable gradient-based learning and exact permutation matrices at the inference time. While a reconstructed adjacency matrix $A_{P'}$ can be different from the input A_P, the decoder always produces a valid permutation matrix. Therefore, the decoder is guaranteed to generate adjacency matrices that represent the same structure as the given graph.

While both Sinkhorn [161] and SoftSort [180] operators produce a permutation matrix, they have several differences in terms of their computational complexity and the mathematical characteristic of produced "soft" permutation matrices. We compare two decoder designs in the evaluation to investigate their differences in building a generative model of matrix reordering.

4.2.4 Training

The parameters of the encoder and the decoder are trained to minimize the reconstruction loss L_X and the variational loss L_Z, as our architecture is a VAE [134]. For the variational loss L_Z, we use the sliced-Wasserstein distance [139] described in Equation 3.3, like our generative model for node-link diagrams (Section 3.2.5).

The reconstruction loss L_X measures the difference between the input matrix reordering $A_P = PAP^{\mathsf{T}}$ and its reconstruction $A_{P'} = P'AP'^{\mathsf{T}}$, where A is the original adjacency matrix, P is the permutation matrix that represents the input node ordering, and P' is the permutation matrix produced by the decoder. We don't consider the

difference between the input node ordering P and the generated node permutation P' since there can be different orderings that produce the same matrix reordering of the input adjacency matrix, as we discussed in Section 4.2.2. We directly compare the difference between the corresponding elements of A_P and $A_{P'}$. For example, we can use the sum of the differences as the reconstruction loss: $L_X = \sum_{i,j} \|(A_P)_{i,j} - (A_{P'})_{i,j}\|$. In addition, since the adjacency matrix of a simple graph is a binary matrix, the binary cross-entropy loss can be used, which is widely used for training a binary classification model: $L_X = \sum_{i,j} -\Big((A_P)_{i,j} \cdot \log(A_{P'})_{i,j} + \big(1 - (A_P)_{i,j}\big) \cdot \log\big(1 - \log(A_{P'})_{i,j}\big)\Big)$.

4.3 Evaluation

This section describes the evaluation of the generative model introduced in this chapter. The primary goal is to validate whether the model can learn to produce diverse matrix reorderings of the given graphs, not just memorizing the training samples. We perform quantitative and qualitative evaluations of the generative model for reconstructing unseen matrix reorderings (i.e., the test dataset).

4.3.1 Datasets

This evaluation uses eight real-world graphs and around ten thousand orderings per graph (Table 4.1). For each graph, we first collected 13,500 matrix reorderings using 25 distance measures between nodes (Table 4.2), 27 matrix reordering methods (Table 4.3), and 20 random initial orderings.

The 25 distance measures consist of the shortest-path distances between nodes, 12 distance measures (Table 4.2) based on the adjacency matrix (a row of adjacency matrix represents a node), and the 12 distance measures based on the adjacency matrix with self-loops (i.e., the diagonal elements of the matrix is 1). The 27 matrix reordering methods (Table 4.3) are selected from the R package seriation [102], which provides publicly available, robust implementations of multiple matrix reordering methods. Although other methods are available in the package, these methods could not compute reorderings of all the eight graphs in a reasonable amount of time (24 hours). The 20 random initial node orderings are used as some matrix reordering methods produce

Name	Type	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	$\|A_P\|$	t	Source
karate	Social network	34	78	2.29	7,964	3.27	[144, 241]
cat	Brain network	52	515	9.90	9,060	4.49	[194]
gd96c	Synthetic	65	125	1.92	10,973	6.65	[177]
macaque	Brain network	71	438	6.17	9,682	7.16	[240]
polbooks	Co-purchase	105	441	4.2	10,303	11.15	[144]
collab	Collaboration	129	847	6.57	8,420	13.68	[236]
ego-fb	Social network	148	1,692	11.43	10,193	16.15	[150, 156]
jazz	Collaboration	198	2,742	13.85	10,212	27.58	[89, 144]

Table 4.1. The eight graphs used in the evaluation. $|V|$: the number of nodes, $|E|$: the number of edges, $|A_P|$: the number of unique reorderings (Section 4.3.1), t: the time it took to collect the training dataset in minutes

different results depending on the initial node ordering of the given graph (Figure 4.5).

A reordering can be the same as the reverse of another. Therefore, a reordering is reversed if its reverse has a higher similarity with a reference reordering—a Spectral reordering is used in this evaluation.

For the reconstruction evaluation, we perform 5-fold cross-validations, where 80% reorderings of a graph are used for training, and 20% reorderings of the graph are used for testing. To properly evaluate the generalization capability of a model, the reorderings in the test set should not be used for training the model. Thus, we extract unique reordered adjacency matrices from the 13,500 reorderings of a graph. The numbers of unique matrix reorderings and the time it took to collect the training dataset of each graph are shown in Table 4.1. Extracting unique adjacency matrices is required only for the evaluation purpose; it is optional for using our technique in practice.

4.3.2 Architectures and Configurations

We compare four different architectures, which are the combinations of two encoder designs (MLP and Conv) and two decoder designs (Sinkhorn and SoftSort): MLP-Sinkhorn, MLP-SoftSort, Conv-Sinkhorn, and Conv-SoftSort. Figure 4.8 shows the encoders and decoders in detail. The encoder architectures generally follow the ones for images. The MLP encoders' design follows the encoder of a variational autoencoder [134]. Conv encoders' design is similar to the discriminator of a deep convolutional generative

Distance	Definition	Distance	Definition
Euclidean	$\sqrt{\sum (u_i - v_i)^2}$	Kulsinski	$\frac{N_{01} + N_{10} - N_{11} + n}{N_{01} + N_{10} + n}$
Manhattan	$\sum \lvert u_i - v_i \rvert$	Rogers-Tanimoto	$\frac{2\,(N_{01} + N_{10})}{N_{00} + N_{11} + 2\,(N_{01} + N_{10})}$
Cosine	$1 - \frac{u \cdot v}{\lVert u \rVert_2 \lVert v \rVert_2}$	Russell-Rao	$\frac{n - N_{11}}{n}$
Dice	$\frac{N_{01} + N_{10}}{2N_{11} + N_{01} + N_{10}}$	Sokal-Michener	$\frac{2\,(N_{01} + N_{10})}{2\,(N_{00} + N_{11}) + 2\,(N_{01} + N_{10})}$
Hamming	$\frac{N_{01} + N_{10}}{n}$	Sokal-Sneath	$\frac{2\,(N_{01} + N_{10})}{N_{11} + 2\,(N_{01} + N_{10})}$
Jaccard	$\frac{N_{01} + N_{10}}{N_{11} + N_{01} + N_{10}}$	Yule	$\frac{2 \cdot N_{01} \cdot N_{10}}{N_{00} \cdot N_{11} + N_{01} \cdot N_{10}}$

Table 4.2. The 12 metrics used in the evaluation for computing the dissimilarity between a pair of nodes. N_{ij} is the number of occurrences of $u_k = i$ and $v_k = j$, and n is the number of nodes. In the evaluation, 25 distance measures are derived using these metrics consisting of 12 distance measures based on the original adjacency matrix, 12 distance measures based on the adjacency matrix with self-loops (i.e., the diagonal elements of the matrix is 1), and the shortest-path distances between nodes.

adversarial network [186]. Both decoder architectures produce a permutation matrix, but with different relaxed sorting operators (Sinkhorn [161] and SoftSort [180]). In this evaluation, we selected the SoftSort [180] operator over the NeuralSort operator [94] as they both produce a unimodal row stochastic matrix while the SoftSort operator is more computationally efficient [180].

MLP encoders use four fully-connected layers, where it takes the input reordering A_P—an $n \times n$ matrix—as an n^2-dimensional vector, where n is the number of nodes of the given graph. Each layer learns a more compressed representation of the input A_P by gradually reduces the dimensionality of representations. From the first to third layers, the i-th fully-connected layer (FC in Figure 4.8) takes an $\lfloor n/2^{(i-1)} \rfloor^2$-dimensional vector and outputs an $\lfloor n/2^i \rfloor^2$-dimensional vector representation of the input A_P. For example, the first layer takes the n^2-dimensional representation of the input A_P and learns an $\lfloor n/2 \rfloor^2$-dimensional representation. The last layer produces the two-dimensional latent

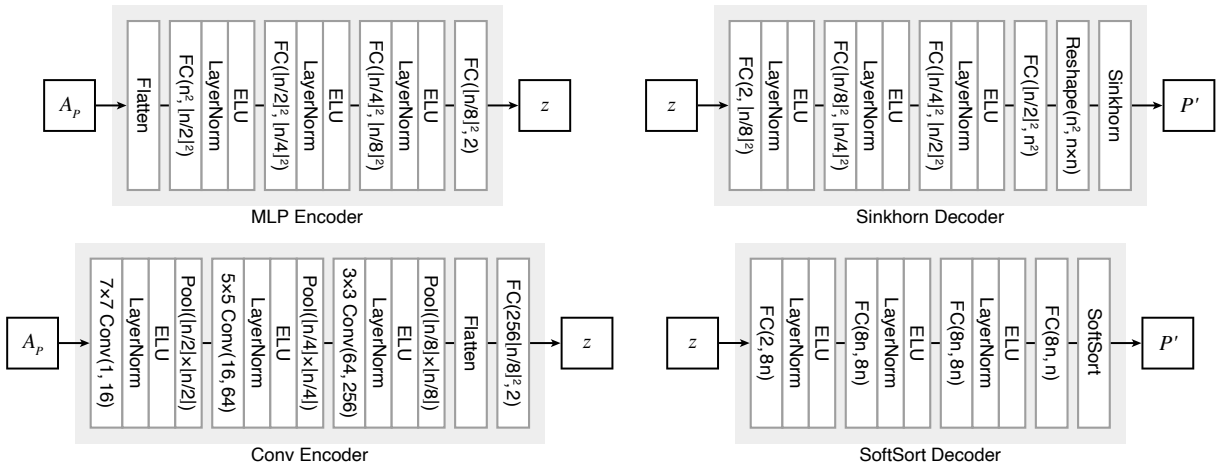


Figure 4.8. The encoder and decoder architectures used in the evaluation. Four architectures are derived based on the combinations of these encoder and decoder designs: MLP-Sinkhorn, MLP-SoftSort, Conv-Sinkhorn, and Conv-SoftSort. FC(m, n) is a fully-connected layer with the input feature size of m and the output feature size of n. k×k Conv(m, n) is a convolution layer with the kernel size of $k \times k$, the input feature size of m, and the output feature size of n. Pool(s) is an adaptive max pooling layer with the target size of s. LayerNorm is the layer normalization [13]. ELU is the exponential linear unit [52].

representation z of the input A_P from the $\lfloor n/8 \rfloor^2$-dimensional representation of it.

Conv encoders use three convolution layers and one fully-connected layer, where the input reordering A_P is considered as an $n \times n \times 1$ tensor. From the first to third layers, the i-th convolution layer (Conv in Figure 4.8) has the kernel size of $(7 - 2(i-1)) \times (7 - 2(i-1))$ and the output feature size of $16 \cdot 4^{(i-1)}$. For example, the second layer has the kernel size of 5 and the output feature size of 64. Each convolution layer is followed by layer normalization [13], ELU [52], and the adaptive max pooling. The i-th adaptive max pooling (Pool in Figure 4.8) has the target size of $\lfloor n/2^i \rfloor \times \lfloor n/2^i \rfloor$. Thus, the first convolution layer takes the input A_P as an $n \times n \times 1$ tensor and outputs an $\lfloor n/2^i \rfloor \times \lfloor n/2^i \rfloor \times 16$ tensor. The last layer computes the two-dimensional latent representation z from the output of the third convolution layer by reshaping the $\lfloor n/8 \rfloor \times \lfloor n/8 \rfloor \times 256$ tensor as a $256\lfloor n/8 \rfloor^2$-dimensional vector.

Sinkhorn decoders use four fully-connected layers and the Sinkhorn operator [161] for computing a permutation matrix. The fully-connected layers of this decoder design is similar to the reverse of the MLP encoder. The first fully-connected layer takes the

two-dimensional latent representation z and outputs an $\lfloor n/8 \rfloor^2$-dimensional vector. From the second to fourth layers, the i-th fully-connected layer takes an $\lfloor n/2^{(5-i)} \rfloor^2$-dimensional vector and outputs an $\lfloor n/2^{(4-i)} \rfloor^2$-dimensional vector. For example, the fourth layer takes an $\lfloor n/2 \rfloor^2$-dimensional vector and outputs an n^2-dimensional vector. The output of the fourth layer is reshaped as a $n \times n$ matrix and fed to the Sinkhorn operator, which produces a "soft" permutation matrix. For computing an exact permutation matrix after training, we apply the Hungarian method [143] on the output of the Sinkhorn operator.

SoftSort decoders use four fully-connected layers and the SoftSort operator [180]. While the input of the Sinkhorn operator is an $n \times n$ matrix, the input of the SoftSort operator is an n-dimensional vector. Thus, the SoftSort decoders can produce a permutation matrix with a significantly fewer number of parameters. In this evaluation, we designed the SoftSort decoders to have $8n$-dimensional hidden representations (see Figure 4.8 for details). The fourth layer takes an $8n$-dimensional vector and outputs an n-dimensional vector, which is passed to the SoftSort operator for computing a permutation matrix. After training, an exact permutation matrix is obtained by applying the row-wise argmax operation to the output of the SoftSort operator.

All the architectures use layer normalization [13] and the exponential linear unit (ELU) [52] on every hidden layer. Batch normalization [122] is commonly used in the neural network architectures for computer vision tasks, especially with convolutional neural networks. However, our pilot study showed that the models with batch normalization often result in numerical instability. The elements of an adjacency matrix are binary, whereas the pixels of a natural image (e.g., landscape, portrait, or an animal) are often 24-bit RGB colors or 8-bit grayscale. Therefore, in our case, batch normalization can lead to numerical instability due to the lack of diverse values of a feature across the instances of a batch.

We use the Adamax optimizer [190] with a learning rate of 0.001 for all the models. In our pilot study, Adamax—a variant of Adam based on infinity norm—showed more stable training than the standard Adam with the L^2 norm. For the reconstruction loss

L_X, we use the binary cross-entropy loss averaged over the elements of the adjacency matrix. We set τ to 1.0 for both the Sinkhorn and SoftSort operators. We train each model for 500 epochs.

Method	Description
ARSA	Minimize the linear seriation using simulated annealing [38].
TSP	Minimize the Hamiltonian path length using a traveling salesperson problem solver [53, 230].
R2E	Rank-two ellipse seriation method [46]
MDS-{Metric, Nonmetric, Angle}	Minimize stress of the linear order using multidimensional scaling (MDS) [112]. Three MDS variants are used.
HC-{Average, Single, Complete, Ward}	Uses a linear order of the leaf nodes in a dendrogram obtained by hierarchical clustering [75]. Each variant uses a different clustering method.
GW-{Average, Single, Complete, Ward}	Hierarchical clustering methods with the leaf-node ordering method developed by Gruvaeus and Wainer [96].
OLO-{Average, Single, Complete, Ward}	Hierarchical clustering methods with the optimal leaf ordering method by Bar-Joseph et al. [16]
VAT	Uses the order of node addition in Prim's algorithm, which finds a minimum spanning tree [28].
Spectral-{$\varnothing$, Norm}	Uses the order of the Fiedler vector [20, 68]. "Spectral" uses the (unnormalized) Laplacian matrix, and "Spectral-Norm" uses the normalized Laplacian matrix.
SPIN-{NH, STS}	Sorting Points Into Neighborhoods [210]. Two variants are used: "SPIN-NH" uses the neighborhood algorithm, and "SPIN-STS" implements the side-to-side algorithm [210].
QAP-{LS, 2Sum, BAR, Inertia}	Formulates reordering as a quadratic assignment problem. Each variant uses different heuristics, namely the linear seriation problem formulation (QAP-LS) [119], the 2-sum problem formulation (QAP-2Sum) [20], the banded anti-Robinson form (QAP-BAR) [73], and the inertia criterion (QAP-Inertia) [43].

Table 4.3. The 27 matrix reordering methods used for producing training data in the experiment. The method names follow the documentation of the R package seration [102].

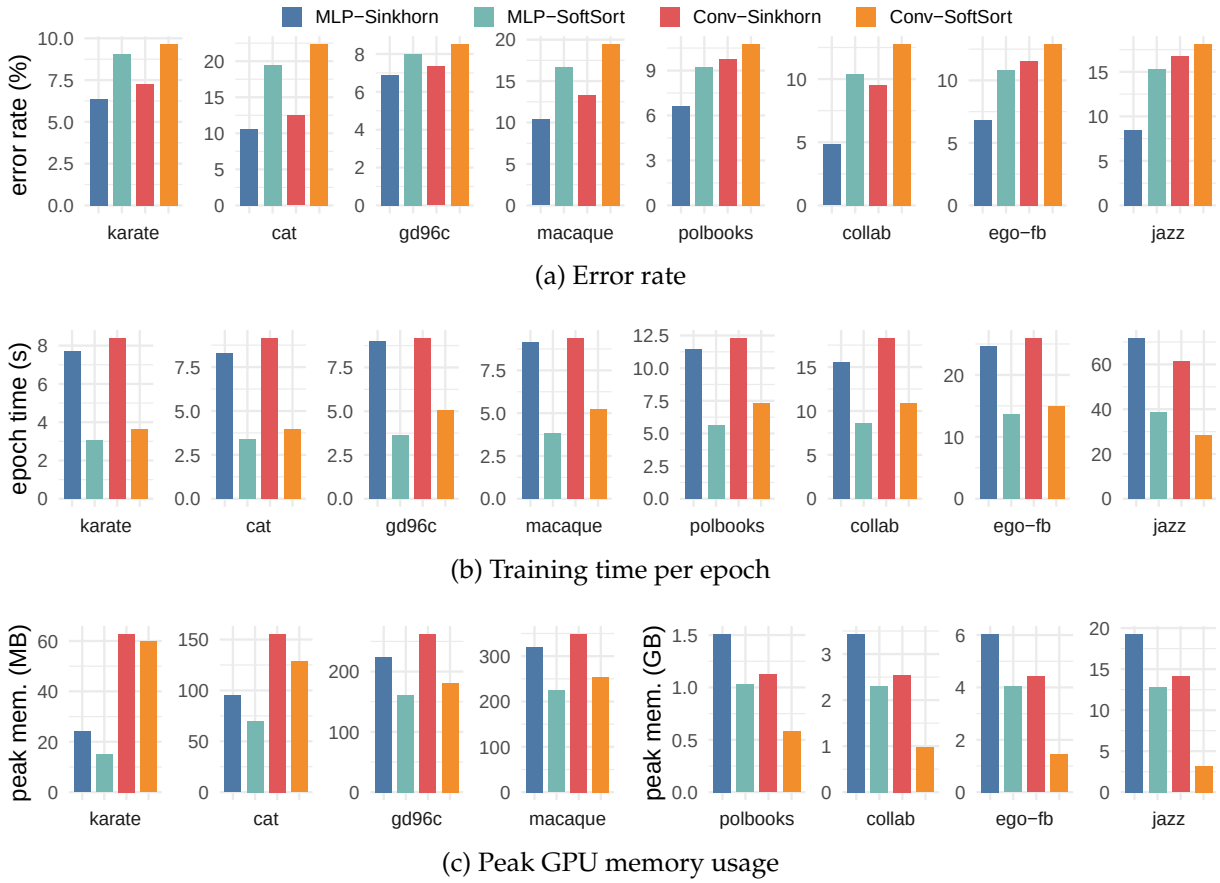
4.3.3 Implementation

We used the R package seration [102] to compute reorderings for the datasets (Section 4.3.1). The models are implemented with PyTorch [174]. To generate the datasets and to run the evaluation, we used a machine with an Intel i7-5960X (8 cores at 3.0 GHz) CPU and an NVIDIA Titan RTX GPU.

4.3.4 Quantitative Evaluation

To quantitatively evaluate the generalization capabilities, we compare four different architectures (Section 4.3.2) for each graph in terms of the average reconstruction error rate of the test dataset—around two thousand reorderings that are not used for training the models. Specifically, a trained model takes an input matrix reordering A_P that is not used for training the model, encodes it to the latent space, and then reconstructs a matrix reordering $A_{P'}$ from the latent representation. The difference between an input matrix reordering A_P and its reconstruction $A_{P'}$ of the test dataset quantifies the generalization capability of a generative model as it estimates how close the model can produce unseen matrix reorderings. For a single reconstruction, the error is measured by the ratio of the different elements of the input matrix A_P and its reconstruction $A_{P'}$: $\left(\sum_{i,j} \mathbb{1}_{(A_P)_{i,j} \neq (A_{P'})_{i,j}}\right) / n^2$. We repeat 5-fold cross-validations ten times and report the mean error rates to reduce the effects of fold assignments. We also compare the four architectures' computational costs based on the training time and the peak GPU memory usage, which measures the GPU memory capacity required for training.

Results Figure 4.9a shows the mean reconstruction error rates of the 10 trials of the 5-fold cross-validations. Overall, MLP-Sinkhorn shows the lowest error rates for all eight graphs. The results show two different rankings of the other three architectures, depending on the graph. For polbooks, ego-fb, and jazz, the models with MLP encoders show lower error rates than the ones using Conv encoders, where the ranking is as follows (from lowest to highest): MLP-Sinkhorn, Conv-Sinkhorn, MLP-SoftSort, and Conv-SoftSort. For the other graphs, the models using Sinkhorn decoders show lower error rates than the ones with SoftSort decoders: MLP-Sinkhorn, Conv-Sinkhorn, MLP-



(a) Error rate

(b) Training time per epoch

(c) Peak GPU memory usage

Figure 4.9. The quantitative evaluation results. The error rates are computed by the ratio of the difference between the input reordering A_P and the reconstructed reordering $A_{P'}$. The standard deviations (or standard errors) are not shown as they are negligible.

SoftSort, and Conv-SoftSort.

Figure 4.9b shows the mean training time per epoch. For all graphs, training the models with SoftSort decoders took a significantly shorter time than those with Sinkhorn decoders. Given the same encoder, training the SoftSort decoders took about half the time it took to train the Sinkhorn decoders, on average, 50.2% shorter. Except for the largest graph (jazz), the MLP encoders took, on average, 13.3% shorter to train than the Conv encoders with the same decoder. Thus, for these graphs, the ranking in terms of the training time is as follows (from shortest to longest): MLP-SoftSort, Conv-SoftSort, MLP-Sinkhorn, and Conv-Sinkhorn. For jazz, training the MLP encoders took, on average, 26.7% longer than the Conv encoders given the same decoder: Conv-SoftSort

(28.4 s), MLP-SoftSort (38.8 s), Conv-Sinkhorn (61.6 s), and MLP-Sinkhorn (71.8 s).

Figure 4.9c shows the peak GPU memory usage for training each model, which measures the required GPU memory capacity. Given the same encoder, the models using SoftSort decoders use, on average, 36.9% less GPU memory than those using Sinkhorn decoders. For the graphs with less than 100 nodes (karate, cat, gd96c, and macaque), the MLP encoders use less GPU memory than the Conv encoders with the same decoder. The difference decreases as the number of nodes increases; it becomes the opposite for the other graphs with more than 100 nodes (polbooks, collab, ego-fb, and jazz), where the Conv encoders use less GPU memory than the MLP encoders given the same decoder.

Discussion For many computer vision tasks, including image generation [129, 186], convolutional neural networks (CNNs) generally outperform fully-connected neural networks (i.e., multilayer perceptrons), even though a CNN often has a significantly fewer number of trainable parameters. However, in this evaluation, the MLP encoders consistently show lower error rates than the Conv encoders when using the same decoder. This difference suggests that matrix reordering requires capturing global patterns of the *images* (i.e., reordered adjacency matrices), which is what CNNs are not good at as they process a local neighborhood at a time.

While the MLP encoders are able to capture global patterns, they are not scalable in terms of the number of nodes, as shown in the GPU memory usage (Figure 4.9c). Thus, a memory-efficient architecture is necessary to support larger graphs. For images, several non-local modules have been introduced to understand global image structures by capturing long-range dependencies between pixels [188, 222, 243]. Although general non-local modules also consume a significant amount of memory, there have been some attempts to simplify non-local modules for images [42, 195] to reduce memory usage. Moreover, we believe taking the sparsity of real-world networks (i.e., an adjacency matrix is often a sparse matrix) [17] into account could drastically decrease memory consumption.

The Sinkhorn decoders show lower error rates than the SoftSort decoders given the

same encoder. This could be simply because the Sinkhorn decoders have more trainable parameters than the SoftSort decoders. However, the experiments of Grover et al. [94] have shown that the NeuralSort operator [94]—which shares many characteristics with the SoftSort operator [180]—outperforms the Sinkhorn operator for sorting tasks, although the models using the NeuralSort operator have a significantly smaller number of parameters than those with the Sinkhorn operator. While both the Sinkhorn and SoftSort operators produce permutation matrices, the Sinkhorn operator is designed for a general matching problem (e.g., align, canonicalize, and sort), whereas the SoftSort (and NeuralSort) operator is designed specifically for sorting the input values (i.e., argsort). Our encoder-decoder architecture and loss function are designed more like a matching problem, where the goal is to produce a permutation matrix that *matches* the input matrix reordering. This design difference can affect the flow of gradients and thus causes the performance differences.

4.3.5 Qualitative Results

Overall, the MLP-Sinkhorn models show the lowest error rates for all the graphs (Section 4.3.4). Figure 4.10 shows several test reconstructions of five different graphs (cat, gd96c, macaque, polbooks, and jazz) using the MLP-Sinkhorn models. The first three rows of reconstructions of Figure 4.10 demonstrate that the MLP-Sinkhorn models can produce diverse styles of matrix reorderings of the given graph. Especially, the reconstructions in the first row are exact reconstructions: the models produce unseen matrix reorderings without any difference. Although exact test reconstructions do not guarantee that the model generates high-quality samples of a generative model, it does demonstrate the generalization capability of our models.

Further investigation of exact reconstructions also revealed that our architectures and loss function are indeed permutation equivariant (Section 4.2.2 and Section 4.2.3). For the karate, collab, ego-fb, and jazz graphs, 99.5% of exact reconstruction cases have the generated permutation matrix P' that is not the same as the permutation matrix of the input matrix reordering P. However, this ratio depends on the number of automorphisms of a graph. For other graphs, all the generated permutation matrix P' is

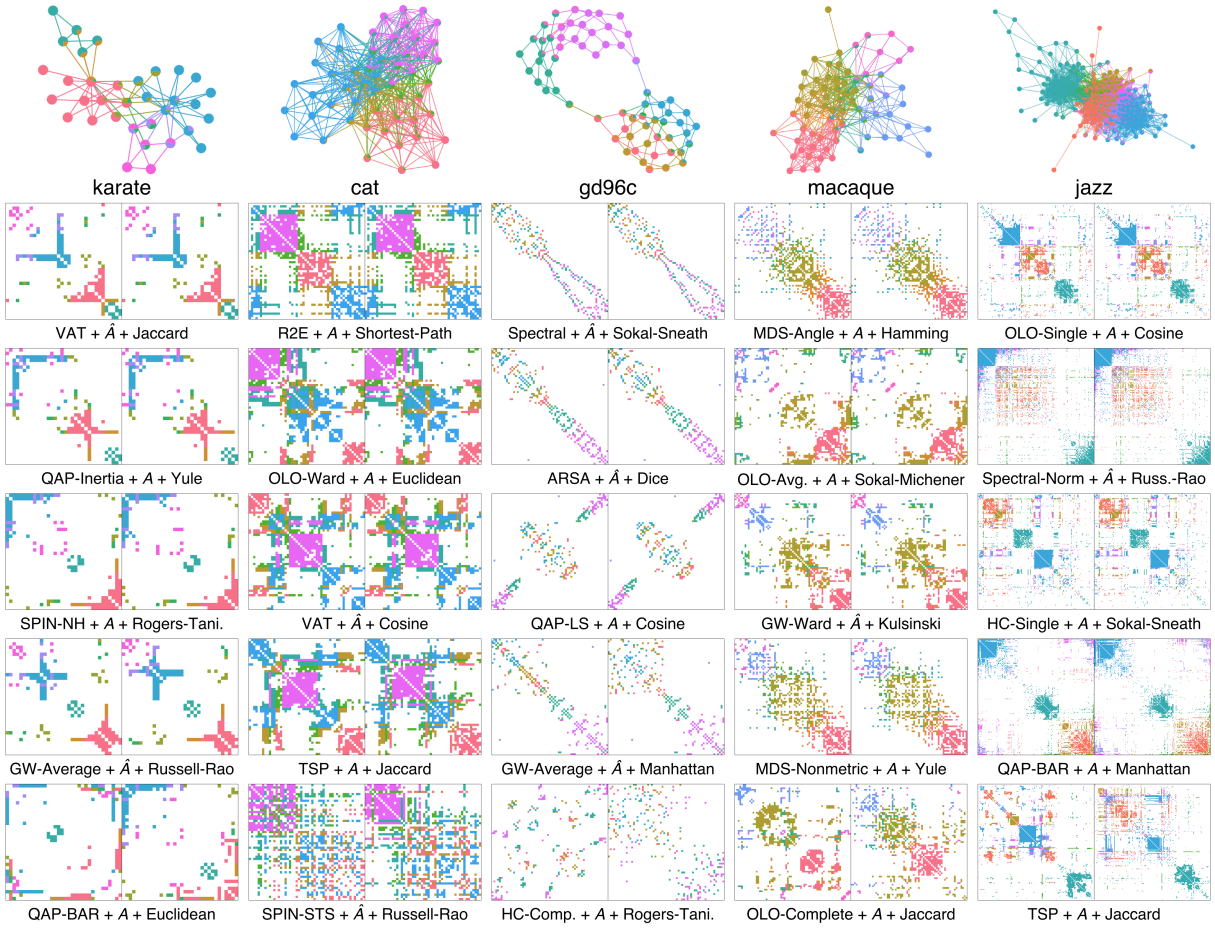


Figure 4.10. Test reconstruction results of the five graphs using the MLP-Sinkhorn models. For each pair of matrix reorderings, the left is test input, and the right is the reconstructed matrix reordering. The label of a pair describes how the input matrix reordering is created: matrix reordering method (Table 4.3), the type of the adjacency matrix (A: the original adjacency matrix, $\hat{A}$: the adjacency matrix with self-loop), and the distance metric (Table 4.2). The results are discussed in detail in Section 4.3.5.

the same as the permutation matrix of the input matrix reordering P.

The bottom two rows of Figure 4.10 shows a number of reconstructions with higher losses (i.e., higher error rates). However, the reconstructions in the fourth row are perceptually similar, although they have relatively higher losses than the first three rows. These results show a limitation of our loss function: the per-cell loss can disagree with the human's perceptual similarity, which is a known issue in other computer vision tasks [244]. We believe a perceptual loss function [244] can be used for improving the quality of the generated samples.

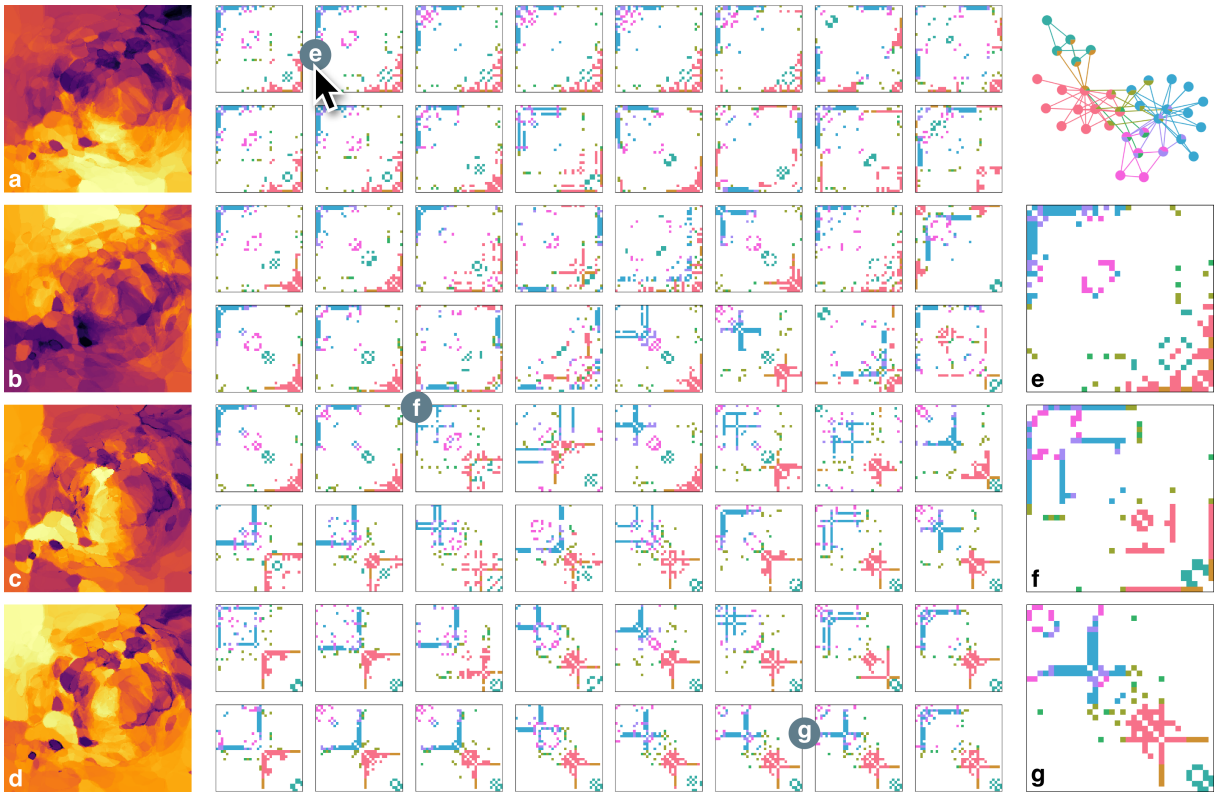


Figure 4.11. The latent space of matrix reorderings for the karate graph learned by a MLP-Sinkhorn model. The grid of samples (center) is generated by decoding 8×8 latent variables z in $[-1, 1]^2$. Users can intuitively explore diverse matrix visualizations and select one they want by simply pointing to a location in the latent space (e–g). The heatmaps (a–d) show four quality metrics of 256×256 matrix reorderings: (a) A (the original adjacency matrix) + Shortest-path distance (distance measure) + the number of anti-Robinson events (AR), (b) A + Hamming + AR, (c) A + Euclidean + Correlation, and (d) $\hat{A}$ (the adjacency matrix with self-loops) + Euclidean + Correlation. The brighter colors represent "better" values in terms of the corresponding quality metric [102]. See Section 4.3.5 for the detailed discussion.

4.4 Usage Scenario

We discuss how a learned latent space can support users in designing effective matrix visualizations of the given graph. Browsing multiple matrix reorderings and selecting the desired one from an unsorted list often lead to tedious trials and errors. Since we train a model to construct a 2D latent space, a grid of generated samples can be created to show several representative matrix reorderings of the given graph that the model learned. Figure 4.11 and Figure 4.12 show the latent spaces of matrix reorderings for the karate, cat, and macaque graphs learned by MLP-Sinkhorn models. The sample grids

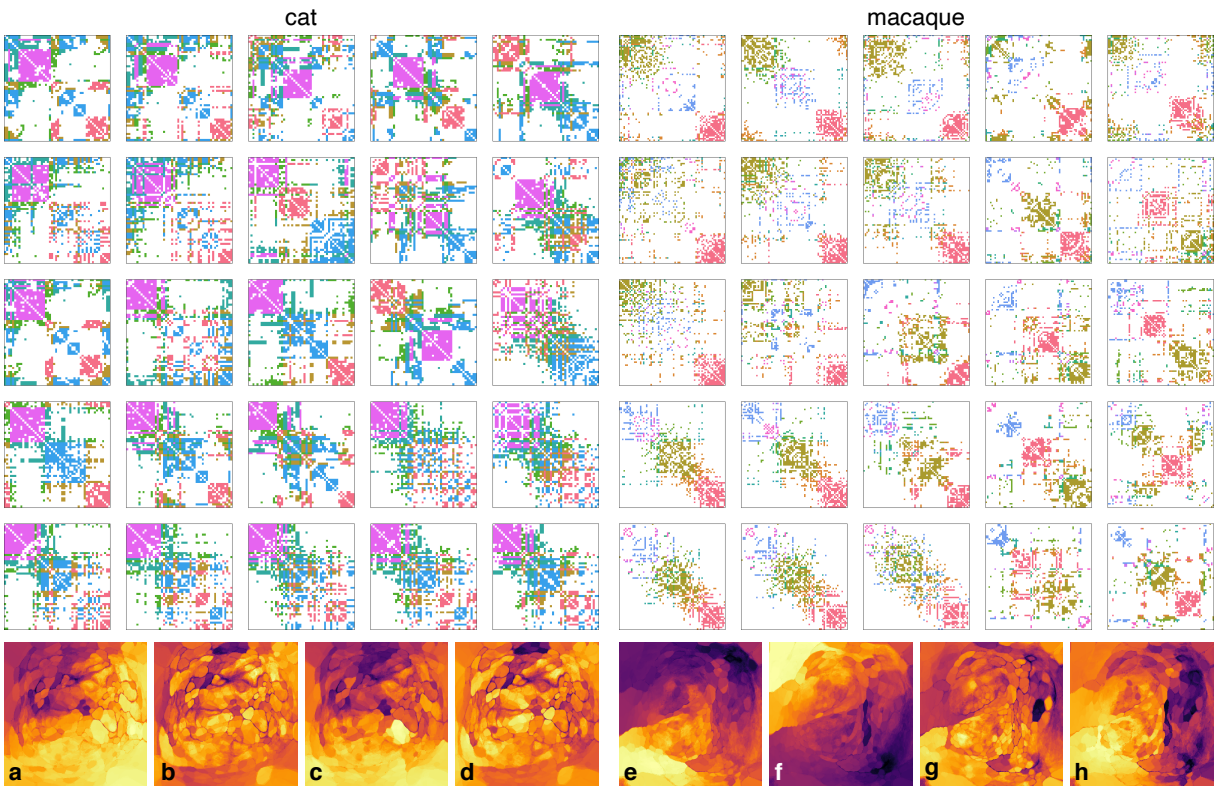


Figure 4.12. The latent spaces of matrix reorderings for the cat and macaque graphs learned by MLP-Sinkhorn models. The sample grids are generated by decoding 5×5 latent variables z in $[-1, 1]^2$. The heatmaps (a–h) show several quality metrics of 256×256 matrix reorderings in $[-1, 1]^2$: (a) A (the original adjacency matrix) + Shortest-path distance (distance measure) + the number of anti-Robinson events (AR), (b) A + Shortest-path distance + the closeness to the banded anti-Robinson form (BAR), (c) A + Yule + AR, (d) A + Yule + BAR, (e) A + Shortest-path distance + AR, (f) A + Hamming + AR, (g) $\hat{A}$ (the adjacency matrix with self-loops) + Cosine + BAR, and (h) A + Yule + Correlation.

provide an organized overview of diverse matrix reorderings of the given graph, where similar matrix reorderings are placed close to each other in the latent space.

Using a sample grid as a *map* of the latent space of diverse matrix reorderings, we build a what-you-see-is-what-you-get (WYSIWYG) interface, where users can intuitively set the latent variable z to produce different matrix reorderings and select one they want by simply pointing to a location in the latent space. For example, Figure 4.11 shows a grid of 8×8 samples throughout the latent space in $[-1, 1]^2$, where the bottom left-corner is $(-1, -1)$ and the top-right corner is $(1, 1)$. When the user points the location (e) in the latent space, the latent variable z is set to $(-0.746, 0.873)$. Then, the model

produces the corresponding reordering shown in Figure 4.11e on the right. Users can effortlessly produce different matrix reorderings by pointing different locations. Pointing the location (f) generates a matrix reordering that highlights the highly-connected nodes of the graph: the blue and red cells that form lines in Figure 4.11f on the right. From the location (g) in the latent space, users can produce a matrix reordering that can show a pattern that is not obvious in the other reorderings (e and f): the olive cells are the edges that connect the blue and red clusters.

Since a learned latent space is $\mathbb{R}^2$, we can create sample grids with different levels of granularity. For instance, a grid of a smaller number of samples can be used for showing more details of the generated adjacency matrices of a larger graph in a limited display area, as shown in Figure 4.12. In addition, we can visualize a latent space as a heatmap with a large number of samples to support experts in matrix reordering (e.g., algorithm designer).

Multiple quality metrics have been introduced to measure the "goodness" of a matrix reordering [102]. Moreover, we can define different distance measures between the nodes of a graph (Section 4.2.1). Thus, finding the quality metric and distance measure that can quantify the desired quality of a matrix reordering for the given graph and analysis goal is not a trivial task.

Figure 4.11a–d and Figure 4.12a–h show several heatmaps that represent quality metrics of 256×256 matrix reorderings of the corresponding graph, where the samples from the brighter areas have "better" values in terms of the definition of each metric (e.g., lower loss values or higher merit values [102]). For example, Figure 4.11a is a map of the number of anti-Robinson events [46] using the shortest-path distances between nodes, where the areas with brighter color have a smaller number of anti-Robinson events. This metric could find a matrix reordering like Figure 4.11g, which can better reveal the olive edges between the blue and red clusters.

By visualizing the latent space with quality metrics, experts in matrix reordering can understand what patterns the given metric can capture or not with diverse and concrete samples. For example, we can clearly see the differences between (1) different quality

metrics (Figure 4.12a and b, and Figure 4.12c and d), (2) different distance measures (Figure 4.11a and b, and Figure 4.11c and d), and (3) different combinations of quality metrics and distance measures (Figure 4.11e–h). Thus, these heatmaps can support the process of designing a new quality metric or distance measure.

In summary, the learned latent spaces support users in designing effective matrix visualizations of a graph by providing a map of diverse matrix reorderings.

4.5 Conclusion

Representing a graph as an adjacency matrix is also a popular approach to graph visualization in addition to using a node-link diagram. However, different node orderings can reveal varying structural characteristics of the same graph. Therefore, finding a "good" node ordering is an important step for visualizing a graph as adjacency matrix views. Users generally rely on a trial-and-error effort to find a good node ordering.

In this chapter, we have introduced a deep generative model that can learn a latent space of matrix reorderings of the given graph. Users can use the learned latent space as a WYSIWYG interface, where they can effortlessly explore diverse matrix reorderings and select the desired one for the purpose of visualization. To achieve this, we have designed a novel neural network architecture to address unique challenges in learning such a deep generative model.

A graph can be visualized in many different ways, where each visualization can highlight different characteristics of the same graph. We hope this chapter and Chapter 3 encourage others to join a new area of graph visualization study, where the goal is to design a method that produces diverse visualizations of the same graph, not just a single visualization that follows only a certain set of heuristics.

Chapter 5
Designing Graph Visualization for Immersive Environments

One of the key freedoms of information visualization design is the opportunity to map arbitrary information to screen space at will, allowing for great control over how the data is presented. As most displays or representations are two-dimensional, most information visualization techniques limit themselves to two-dimensional mappings. That is, information visualization approaches traditionally eschew 3D techniques; 3D visualizations projected onto 2D displays often lead to occlusion and clutter issues. Better 2D representations generally avoid this problem. However, such a limitation is only applicable to 2D displays. The onset of ubiquitous, consumer-level stereoscopic displays has prompted questions of the potential effectiveness of 3D for information visualization techniques.

Whereas 3D displays used to be rare and cost-prohibitive, recent advances have made them ubiquitous enough that they can be assumed to be available for general visualization approaches. The most common type of these displays currently exists in the form of standard displays paired with eyewear that reveals different images per eye, which is either active (where an LCD over each eye alternates views in sync with the display) or passive (polarization of light differs per eye). However, even these displays limit the user's view to a small rectangular window encapsulated by the display area.

While numerous information visualization techniques exist to visualize large data with limited display space, screen area still presents a concrete limit, particularly since the human eye is capable of utilizing a much larger field of view (FOV).

Larger displays such as powerwall displays or CAVE systems [80] aim to make use of the full range of human vision, and enable nigh unlimited immersive techniques, but these systems are often too expensive and bulky spacewise to build, as they require a large number of displays, and massive amounts of processing power to drive them all.

Head-mounted displays (HMDs) have recently become a popular alternative. In recent years, a number of HMDs are released or announced to be released including Oculus Rift [1], HTC Vive [2], Sony PlayStation VR [3], Google Cardboard [4], and Microsoft Hololens [5]. With these devices, a small, but high resolution display is placed directly in front of the user's eyes, such that each eye's view can be tightly controlled. Thus, it is unnecessary to show the entire scene; only the user's view needs to be rendered. Due to recent rapid improvements in features such as pixel density, high refresh rate, and low latency head tracking, HMDs have quickly become both feasible and affordable. Consumer-grade HMDs are quickly nearing a point where they can be treated as commodity hardware. These devices create immersive environments at a fraction of the cost of large display systems. The ubiquity of such devices enables a multitude of visualization possibilities.

Immersive scientific visualization techniques have been well explored, as 3D environments are ideally suited for 3D immersion. As such, scientific visualization has been a driving force behind the development of virtual reality systems. However, investigation into the applicability of such immersive environments for non-spatial data has been extremely sparse. Even the concept of employing stereoscopy for non-spatial data has had limited exploration [36, 159]. In fact, previous research has found that stereoscopic representations are particularly helpful for graph visualization in certain tasks [11, 93, 225, 226], though these approaches limit themselves to traditional rectangular displays.

In this chapter, we present a novel study of layout, rendering, and interaction

methods for immersive graph visualization. Specifically, the primary contributions of our work are:

- New layout methods and the edge routing techniques specifically designed for immersive environments.
- Rendering techniques for enhancing clarity and highlighting techniques to better support interaction with the graph visualization.
- A user study for comparing different layout methods using a number of common graph exploration tasks in an immersive environment.

The results of the user study show that traditional 2D graph visualization are ill suited for immersive environments. Methods specifically designed for immersive graph visualization like ours are in need. In this manner, we have not only established an improved understanding of the effectiveness of the immersive graph visualization, but also posited general guidelines for future immersive visualization approaches.

5.1 Related Work

Virtual reality and stereoscopic techniques have a long history of being used in scientific visualization [37, 39, 213], applied to GIS data [26], volume data [229], and medical imaging data [164]. When the data has a 3D spatial attribute, such techniques are a natural fit.

The use of 3D in information visualization has been demonstrated for a long time [36, 208]. However, most existing 3D information visualizations are displayed on 2D displays with the monocular depth cues such as perspective, occlusions, motion parallax, and shading [58, 223]. Stereoscopic 3D displays provide one additional important depth cue: *binocular disparity*. Stereoscopy has been shown to be beneficial for some information visualization tasks [159]. Notably, stereoscopy has been shown in multiple user studies to be effective for graph visualization tasks [11, 93, 225, 226].

The earliest study on using stereoscopy for graph visualization was done by Ware and Franck [224]. They considered general low-level tasks for graph analysis such as identifying paths between two highlighted nodes, and found that stereoscopy with head

tracking outperforms those in 2D condition. About ten years later, Ware and Mitchell [225,226] conducted similar studies but in a higher resolution display. They found that participants showed less error rates with both motion parallax and stereoscopic depth cues, and stereoscopic viewings showed faster response times, regardless of the motion parallax. A more recent study done by Alper et al. [11] showed the effectiveness of stereoscopic highlighting techniques for 2D graph layout. While there was no significant difference between the stereoscopic highlighting and static visual highlight (color), participants performed better when both highlighting were used together. Halpin et al. [104] used an immersive system, but they still used a standard Euclidean layout and employed the stereoscopy just for highlighting. Barahimi and Wismath [18] used HMD to view a 3D layout. In most of these studies, the graph is generally looked at from outside so that they often requires lots of viewpoint navigation and cluttered.

To keep the user's viewpoint equidistant to most of the display area, immersive environments can be modeled very naturally with a spherical or a cylindrical paradigm, with the user at the center. While there are lots of 3D graph layout approaches [110], few graph layouts work in a spherical space [120,138,168,205,233] because most existing graph layouts perform their calculations in the standard Euclidean space. Kobourov and Wampler [138] introduced a force-directed method to calculate a graph layout in an arbitrary Riemannian geometry. Hyperbolic graph layout works in a spherical space [167], but nodes/edges would go through the user's view in the center of the sphere. Wu and Takatsuka [233] used a self-organizing map to place a small graph on a sphere, but this is based on multidimensional node properties. While these methods work in the spherical space, they are not designed to be looked at from the inside of the sphere. In this chapter, we introduce layout and interaction techniques specifically designed for immersive environments.

In graph visualization, one challenge is how to effectively handle large number of edges. Edge bundling techniques [77,113,114] are common methods for routing edges to avoid clutter. While 3D layouts can eliminate many edge crossings by utilizing the additional dimension, conveying the depth of the edges presents an additional challenge.

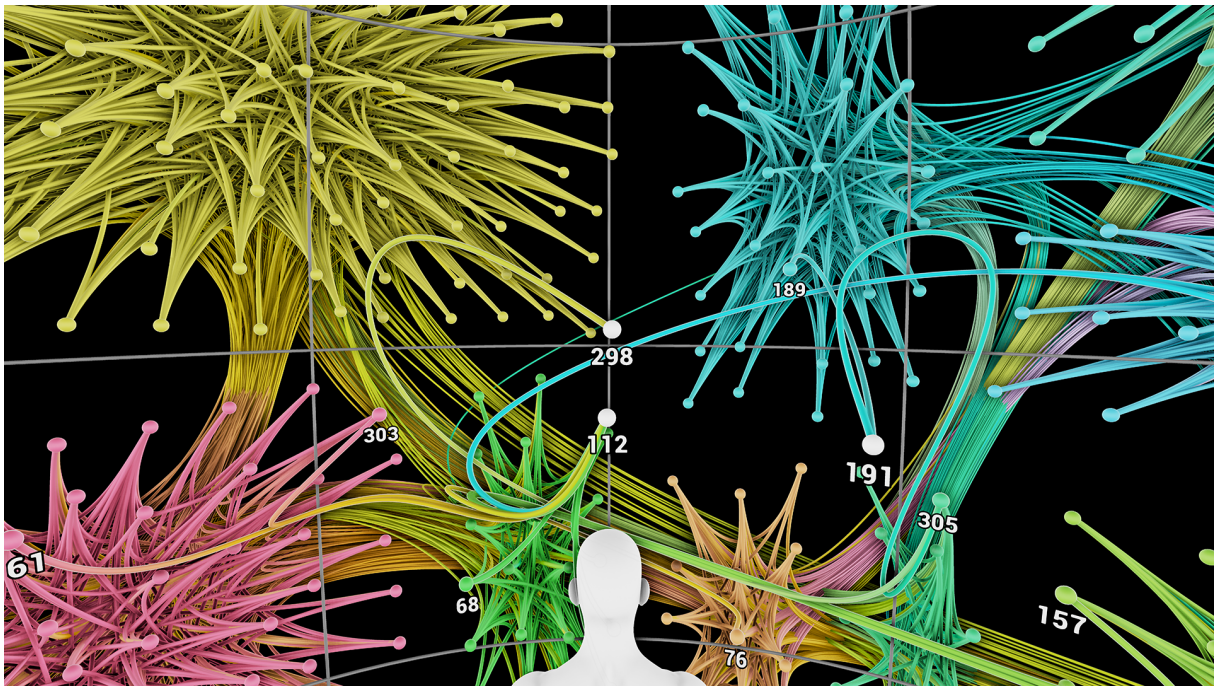


Figure 5.1. The viewer is placed at the center of the sphere, on which the graph is laid out.

The studies of vector field visualization and diffuse tensor imaging techniques have explored this same problem. One basic technique is to illuminate the lines [15, 154]. It has also been shown adding halos can help enhance perception of relative depth between lines [78, 79, 152], though halos do not work for dense lines. To add the effect of global illumination, ambient occlusion may be used as done by LineAO [74]. We adopted a simplified version of LineAO [74] to emphasize the structure of a graph without compromising the frame rate.

5.2 Methods

In an immersive environment, the visible scene is arranged omnidirectionally around the user's viewpoint. As much of the scene will be out of the user's view at any time, this naturally evokes exploration through head motion. Performing such navigation comfortably is often limited by the range of motion of the user's neck, which is often relatively fixed positionally, but flexible angularly (e.g., in a seated usage). Therefore,

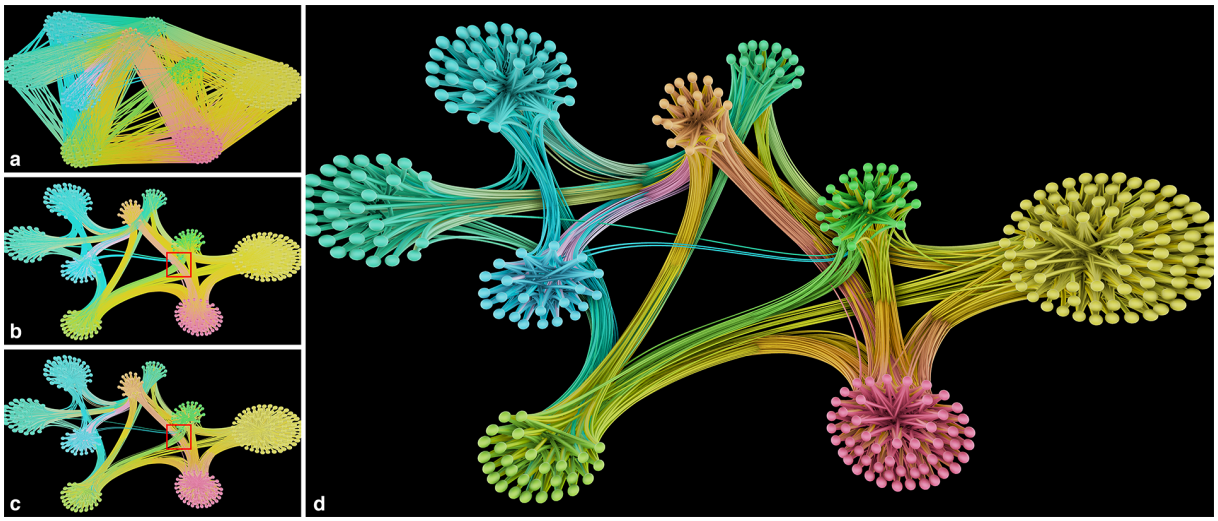
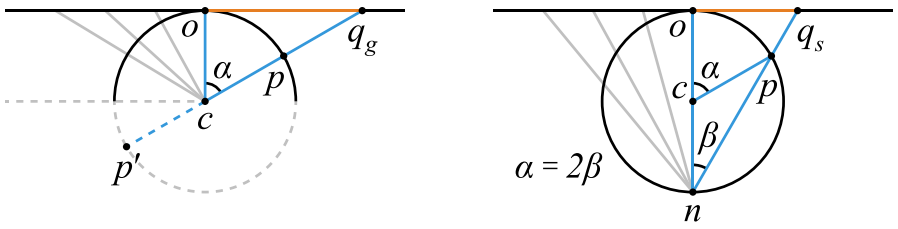


Figure 5.2. Layout and rendering strategies. (a) place nodes on the surface of a sphere; (b) employing spherical edge bundling; (c) adding depth routing; (d) adding illumination. The detail views of the red rectangle can be found in Figure 5.8.

visibility of the scene from the perspective of the user's viewpoint is vital. However, traditional 3D graph layout often requires lots of spatial navigation in order for the user to find viewpoints that are good for perceiving depth and comprehending the structure of the graph [11]. To improve on this, it is important to find or create an ideal viewpoint that does not require the user to perform such heavy navigation [93]. Our approach targets this by using graph layouts on the surface of a sphere and by placing the user's viewpoint at the center of the sphere (Figure 5.1), so that all nodes are equally visible to the user through angular motion alone. In order to reduce occlusions, edges are routed outside the sphere in a bundled fashion (Figure 5.2c). Lastly, as graph rendering involves rendering large numbers of lines, we utilized dense line rendering techniques from scientific visualization works (Figure 5.2d).

5.2.1 Spherical Graph Layout

It is non-trivial to calculate the layout of a graph on the surface of a sphere as it is non-Euclidean space. While several studies [120, 138, 168, 205, 233] introduced methods that can calculate a graph layout on the surface of a sphere, most existing graph layout algorithms are designed for 2D Euclidean space. Thus, while dedicated spherical graph



(a) Gnomonic projection geometry

(b) Stereographic projection geometry

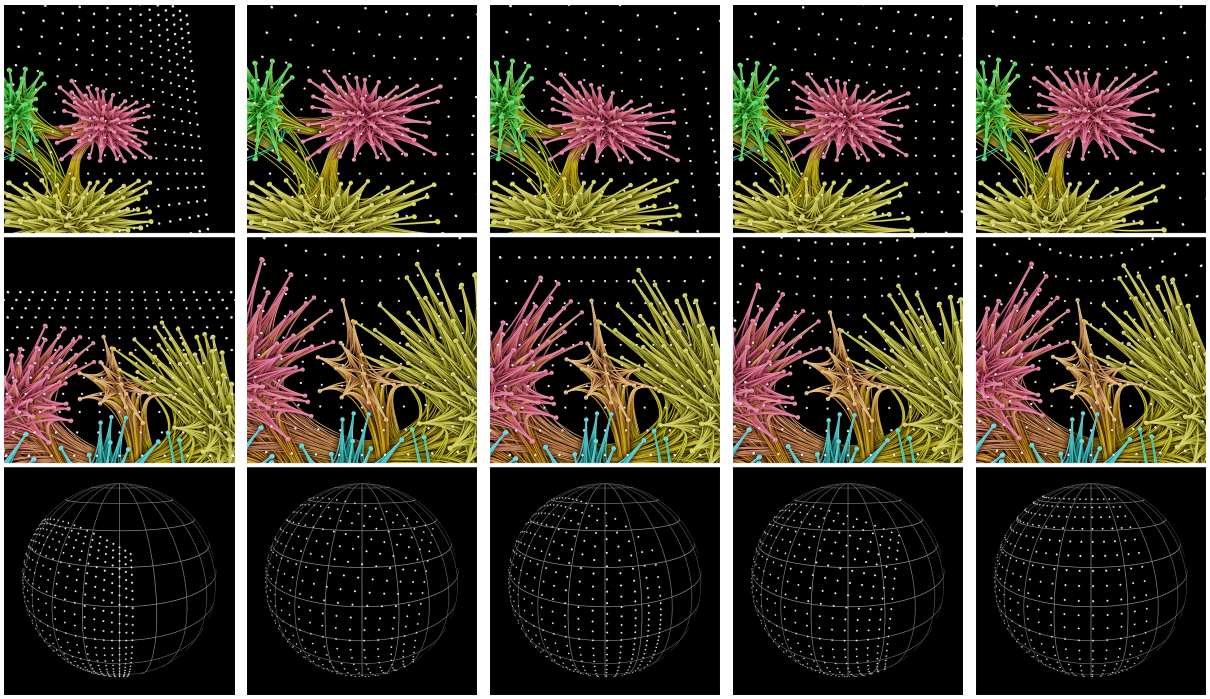
Figure 5.3. Spherical projection geometry. Linear distances in the plane are not linear on the surface of the sphere (gray lines).

layouts are possible, this work focuses on methods of effectively mapping existing 2D graph layout approaches to the surface of a sphere, in order to maximize the utilization of existing graph works.

While the surface of a sphere is also a 2D space, it is impossible to map a 2D Euclidean geometry to the surface of a sphere without distortion. The field of cartography has explored map projections from a sphere to a plane fairly extensively [125]. To map a 2D graph layout to the surface of a sphere, we need a process that calculates the inverse of such map projections (i.e., that maps locations on a plane to locations on the surface of a sphere). Of the numerous kinds of map projections, we focus on gnomonic and stereographic projections, because they provide continuous mappings to arbitrary planes, and have well defined distortions that can be compensated for. We also used spherical coordinates and a cubed sphere as additional ways to map 2D layouts onto the sphere's surface.

5.2.1.1 Gnomonic Projection

In a gnomonic (or rectilinear) projection, a hemisphere is projected from its center radially outward onto a plane that is tangent to the hemisphere at a point. One advantage to this projection is that great circles are projected to straight lines on the plane, i.e., a straight line on the plane is a geodesic arc on the sphere. Gnomonic projection (5.3a) projects a point p on the surface of a sphere from the center c of the sphere to point q_g on a tangent plane of a point o [57]. This projection can only be used to project one hemisphere at a time because it projects antipodal points p and p' to the same point q_g



(a) Direct Gnomonic Projection

(b) Radially Compensated Gnomonic Projection

(c) Independently Compensated Gnomonic Projection

(d) Direct Stereographic Projection

(e) Spherical Coordinates

Figure 5.4. Mapping 2D graph layouts to the surface of a sphere. The upper two rows of images are what the user sees, and the bottom row of images show the mapped grid points on the sphere surface. The top row of images show the corner area of treemap-based layout [165]. The middle row of images show Gosper curve based layout [166] as an example of roughly circular layout.

on the plane.

Let us consider a unit sphere in $\mathbb{R}^3$. There is no distortion at the tangent point o, but distortion increases the further a point is away from o. The geodesic arc op on the sphere corresponds the line segment $\overline{oq_g}$ on the plane. By considering the points o, c, and q_g as a right triangle, the length of $\overline{oq_g}$ is $\tan \alpha$. The result of this is that in a naive, direct mapping the points from the plane to the surface of a sphere, the points further away from o become compacted on the sphere, as can be seen in 5.4a. Therefore, before mapping a point from the plane onto the sphere, we compensate for this by normalizing the point and then distorting them radially according to $q'_g = o + \tan \|\mathbf{u}\| \cdot \hat{\mathbf{u}}$, where $\mathbf{u} = q_g - o$. While this is good at preserving radial distances, one limitation of direct

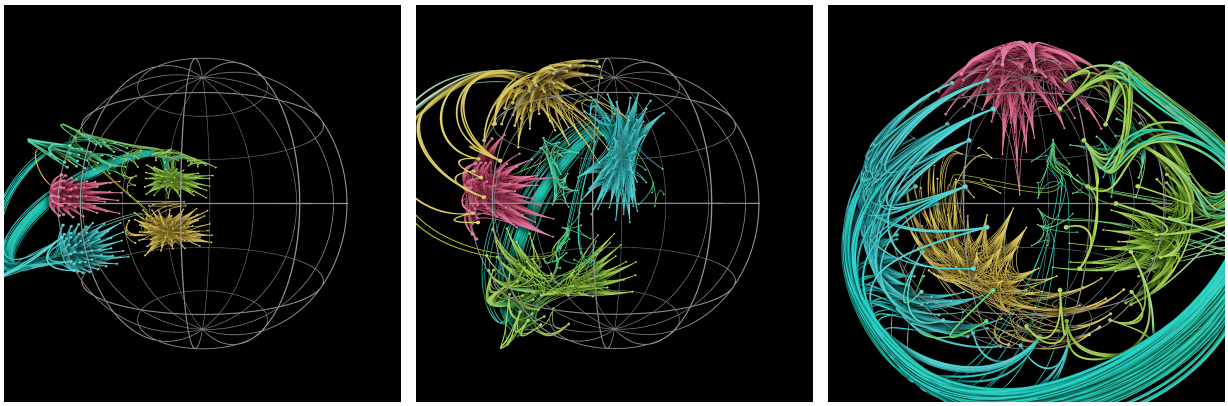
application of this mapping is that it does not preserve angles or straight lines. This becomes especially apparent with rectangular input layouts (such as a treemap-based layout [165]). This angular distortion can be seen in the corners in 5.4b. Also, the FOV available for the graph is limited because corners get wrapped much further around the sphere than the sides, so this mapping is limited by extrema in the layout, such as corners or outliers. Thus, this specific mapping works best with roughly circular layouts. If the layout is more circular and less dependent on straight-line boundaries (as in many force directed layouts or the Gosper curve based layout [166]), the angular preservation is less of an issue.

As a compromise, we also employ the option of warping the x and y dimensions independently as $x' = \tan x$ and $y' = \tan y$ before mapping, as demonstrated in 5.4c, In this manner, corners are not overly extended. Also, horizontal or vertical lines are preserved as perceptually linear arcs along the surface. However, distortions in area will increase as the FOV increases. Notably, the area of regions near the bounds of a rectangular area would shrink to zero as they near the hemisphere boundary, meaning this approach is only really useful for relatively narrow FOV. In practice, this approach is best suited to a limit of 120° FOV for the graph.

5.2.1.2 Stereographic Projection

Stereographic projection [57] projects a point p on the surface of a sphere to the point q_s on the plane tangent to the sphere at point o along a ray originating from the antipodal point n opposite o, as illustrated in 5.3b. Similar to gnomonic projection, stereographic projection offers very predictable and analyzable distortion. Unlike gnomonic projection, stereographic projection is good at preserving angular properties. Notably, circles on the sphere that do not pass through the point n are projected to circles on the plane and vice-versa.

As in gnomonic projection, regularly spaced points in a Euclidean space would be skewed when projected to spherical space, as can be seen in 5.4d. Unsurprisingly, this follows the same tangential law as Gnomonic projection, so the same correctional options are applicable. However, one big difference between gnomonic projection



(a) 120°×67.5° FOV (b) 180° FOV (c) 360° FOV

Figure 5.5. Spherical graph layouts: 2D layouts can be mapped to the sphere with some amount of distortion. Preserving angles with independent axis corrective mapping (a) is appropriate for rigid, rectangular structures, but is limited in field of view (FOV). Radial corrective mapping (b) works well for roughly circular layouts on a hemisphere. For full immersion (c), we use a space filling curve defined on a cubed sphere to cover the entire surface.

and stereographic projection is that the angle β in stereographic projection is half of the angle α in gnomonic projection. Thus, we can either warp the plane radially by $q'_s = o + \tan\frac{\|\mathbf{v}\|}{2} \cdot \hat{\mathbf{v}}$, where $\mathbf{v} = q_s - o$, or warp the x and y dimensions independently as $x' = \tan\frac{x}{2}$ and $y' = \tan\frac{y}{2}$.

5.2.1.3 Spherical Coordinates

Another straightforward method is mapping 2D coordinates (x, y) to spherical coordinates (ϕ, θ). The major difference between this and the projection-based approaches is that in spherical coordinates, all points along the equator are undistorted, but the distortion increases towards the poles. As shown in 5.4e, points near the poles are greatly compacted. Unlike in previous approaches, this distortion is not as simple to account for. However, by limiting the range of the vertical angular axis, the consequences of this distortion can be minimized. While this limits the vertical range of space that is utilized, it still allows the use of up to the full 360 degree horizontal range.

5.2.1.4 Cubed Sphere Mapping

For a completely immersive layout, the nodes should be distributed roughly evenly in all directions, not just on the front hemisphere. One of the best ways we found to do

this was to employ a cubed sphere, where a graph layout is calculated onto a cube, and then each face of the cube is mapped onto the sphere. As the gnomonic projection, the grid cells of each cube face are warped with a tangent function. Because of the rigid boundaries, it is imperative to preserve straight lines, as the interfaces between the faces should be contiguous. Thus, we warp the x and y dimensions independently forming what is commonly referred to as an equiangular cubed sphere [65].

A space-filling curve layout then maps very nicely to this approach, as a space-filling curve can be defined on the surface of a cube as six planar space-filling curves, one each face, that are still contiguous as one single curve [65]. To compute this layout, we break the graph into 6 sections, lay out the nodes for each section in a separate plane, warp their x and y values according to a tangent function, and finally map each plane onto the appropriate face of the cubed sphere. This ensures an even distribution of nodes over the entire surface, while preserving cluster locality.

5.2.2 Field of View Variation

The one thing all the spherical mappings have in common is that there is flexibility in the range of the sphere to utilize. That is, the layout algorithm has control of how much FOV to utilize for the graph.

One important aspect to consider is the range of motion of the user's neck (cervical spine). Particularly when the users seated, they often look forward, and to the left and right without moving their torso. Also, the visual density of a graph should be considered to determine appropriate FOV for the graph. Too wide of an FOV can be overly strenuous to the user, and can hide parts of the graph in regions the user might not even see. Conversely, too narrow FOV can be overly cluttered and less immersive. So it is generally better to limit the FOV of the layout to be within the average range of motion of neck, unless the graph is complex enough to benefit from wide FOV.

There are several criteria to measure this range. We used what is referred to as the active range of motion (the range of movement through which a person can actively move the joint without any assistance). Several studies [6, 239] show that the vertical range of the motion of is roughly 50° upwards (extension) and 60° downwards (flexion),

and the horizontal range left and right are around 80° each. Thus, the maximum range to limit ourselves to should be around 160° horizontal by 100°–110° vertical. We confirmed this during our pilot study, where we found that the users were not comfortable near or over these limits. So, in our evaluation of the effect of FOV within these limits we used layouts at 3 FOVs: with horizontal FOVs of 150°, 120°, and 90°, each with a 16×9 aspect ratio.

5.2.3 Edge Bundling

Edge bundling techniques [77, 113, 114] are frequently used to improve the legibility of dense or complicated node-link diagrams. The main difference between these techniques lies in how the curves of the edges are calculated. These techniques use curves such as Bezier curves, B-splines, or Catmull-Rom splines to route edges along control points. We apply a variant of hierarchical edge bundling [113] to our immersive graph visualization.

Direct application of edge bundling to a spherical layout using computations in 3D Euclidean space would not work well for our purposes, as the bundles would run through the inside of the sphere close to the viewer, obscuring the nodes of the graph. Rather, just like the layout of the nodes, we route the edges around the surface of the sphere instead. In addition, we also route edges away from the outside of the sphere according to the clustering hierarchy, with the edges that has longer path along the clustering hierarchy routed further away from the sphere than short ones, in order to improve visibility when viewed from the inside of the sphere.

5.2.3.1 Spherical Edge Bundling

The first step is to route the edge on the surface of the sphere. That is, given the start point, end point, and a set of intermediate control points, all on the surface of the sphere, we want to compute a cubic B-spline that also lies on the surface. Most spline calculation algorithms are defined using linear interpolation in Euclidean space, such as de Boor's algorithm [62]. In the Euclidean space, a B-spline is calculated using de Boor's algorithm as:

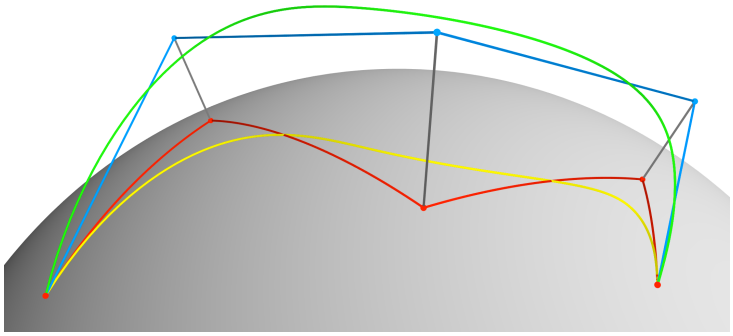


Figure 5.6. Hierarchical edge bundling routes edges with splines that follow the clustering hierarchy. Each control point corresponds with a cluster node in the hierarchy. We compute the edge spline in two stages. A spherical spline (yellow) is calculated with a spherical B-spline according to the control points on the surface (red). Then, we calculate the 1D depth spline according to the height of the corresponding cluster nodes in the clustering hierarchy (blue). Finally, the spherical spline is extended radially (green) by applying the depth spline.

$$
\begin{aligned}
p_i^j(t) &= \begin{cases} \mathrm{LERP}(p_{i-1}^{j-1}(t), p_i^{j-1}(t), r_i^j) & \text{if } j > 0 \\ p_i & \text{if } j = 0 \end{cases} \\
&\text{where } r_i^j = \frac{t - t_i}{t_{i+k-j} - t_i}, \\
\mathrm{LERP}(p_0, p_1, t) &= (1-t)p_0 + tp_1
\end{aligned}
\tag{5.1}
$$

However, linear interpolation in Euclidean space would fail to compute correctly on the surface of a sphere, as would linear interpolation in spherical coordinates, as the surface of a sphere is non-Euclidean. In this chapter, we used a modification of de Boor's algorithm that use *s*pherical *l*inear int*erp*olation (SLERP [200]) instead of Euclidean linear interpolation to calculate splines on the surface of a sphere, which is defined as:

$$
\begin{aligned}
\mathrm{SLERP}(p_0, p_1, t) &= \frac{\sin(1-t)\theta}{\sin\theta} p_0 + \frac{\sin t\theta}{\sin\theta} p_1, \\
&\text{where } \theta = \arccos(p_0 \cdot p_1)
\end{aligned}
\tag{5.2}
$$

Replacing the Euclidean linear interpolation with the spherical linear interpolation

in de Boor's algorithm yields a spherical B-spline:

$$p_i^j(t) = \begin{cases} \text{SLERP}(p_{i-1}^{j-1}(t), p_i^{j-1}(t), r_i^j) & \text{if } j > 0 \\ p_i & \text{if } j = 0 \end{cases} \tag{5.3}$$
$$\text{where } r_i^j = \frac{t - t_i}{t_{i+k-j} - t_i}$$

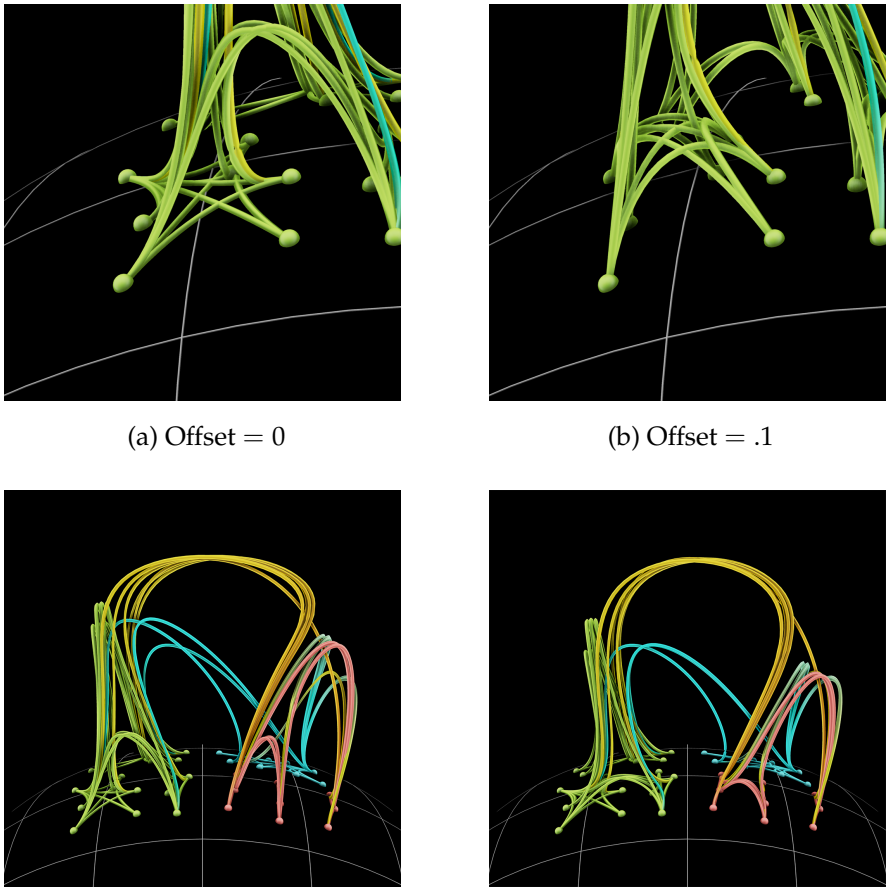
In this manner, the spline is smooth from the perspective of the user in the center of the sphere. While several studies [41, 133] have introduced more complex methods, we found de Boor's algorithm with SLERP to be sufficient, and it is relatively simple and computationally efficient enough to recompute upon interaction. An example of such a spline is shown as the yellow curve in Figure 5.6, which is computed according to the red control points.

5.2.3.2 Depth Routing

Bundling edges using spherical splines improves the legibility of the spherical graph layout equivalent to edge bundling in traditional 2D layouts. But it places all edges at the same distance from the user's viewpoint, which not only causes edge crossings which could be avoidable, but also fails to take advantage of depth perception capabilities of stereoscopy.

So in addition to the spherical spline computation, we also calculate a 1D depth spline to route the edges depthwise by raising the spline off the surface of the sphere by modulating the distance of the sample points with the samples of the depth spline. Each control point corresponds with a node in the clustering hierarchy, so we calculate the depth spline according to the height of the corresponding cluster node (i.e., distance in the cluster hierarchy from the cluster node to its deepest leaf node).

We calculate the distance from the center of the sphere to a each control point of the depth spline u in the clustering hierarchy as $d_u = r(1 + o + s \cdot h_u^p)$, where r is the radius of the sphere, o is an offset value, s is an scale value, p is an exponent value, and h_u is the height of the node u in the hierarchy. The offset value o (5.7a and 5.7b) determines how much deep to route edges between nodes within the same cluster. 5.7c and 5.7d shows the effect of the exponent value p. By varying the offset value o, the scale value s



(a) Offset = 0

(b) Offset = .1

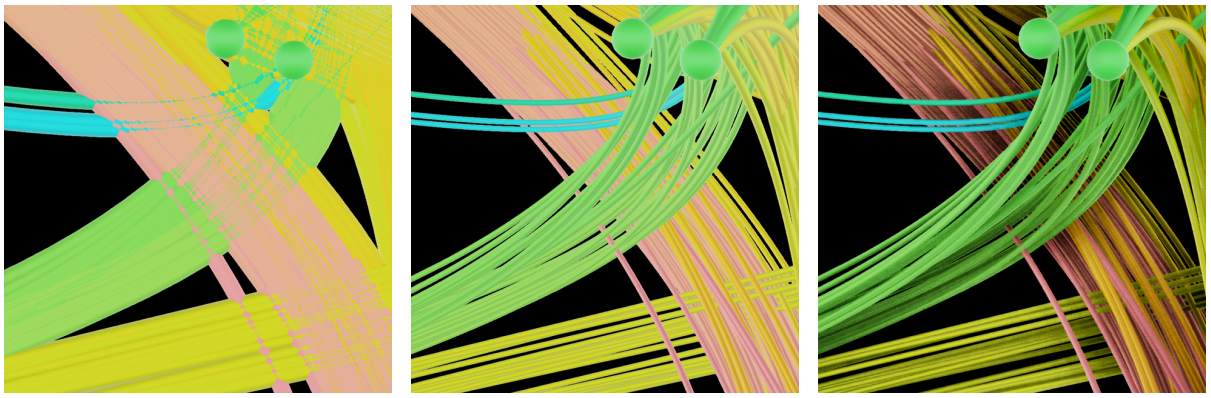
(c) Exponent = 1, Scale = 1

(d) Exponent = 3, Scale = .25

Figure 5.7. Depth routing parameters. By varying the offset, the scale value, and the exponent value, we can fine-tune the edge bundling to improve the clarity of the visualization.

and the exponent value p, we can fine-tune the edge bundling to improve the clarity of the visualization.

In this manner, an edge is routed up the clustering hierarchy, moving further away from the user's viewpoint up to the least common ancestor cluster, and then back towards the user as it traverses down the other side of the clustering hierarchy. The result of this is that the longer, inter-cluster bundles are also the furthest away from the user, while shorter, more intricate structures remain close to the viewpoint. While closer edges occlude other edges from the user's viewpoint, the edges with the depth routing (5.8b) are longer, and bundled, making them easier to perceive. Also, the legibility is improved even further with illumination techniques, as shown in 5.8c.



(a) without depth routing (b) with depth routing (c) with depth routing and illumination

Figure 5.8. Depth routing and rendering techniques.

5.2.4 Rendering Techniques

Rendering dense bundles of lines has been well studied for scientific visualization, such as diffusion tensor imaging or streamline visualization. As such, there are numerous techniques available that could be applied to graph visualization, such as illuminated lines [15, 154], halos [78, 79], or LineAO [74].

LineAO [74] is particularly effective since it is both utile and efficient. It emphasizes both local structure and global structure using multiple hemispheres of different radius for ambient occlusion sampling. To enable maximum frame rate of our implementation, we simplified LineAO to using two hemispheres of two radius (local and global). Figure 5.8 shows our simplified LineAO result.

Standard line rendering produces unintuitive perceptual cues at close distances. Specifically, any strand-like object in the real world will appear larger at close distances and smaller when far away. In order to convey this, simple line rendering is not nearly sufficient. As such, we have to employ tube rendering techniques, so that the width and shading will correspond correctly with depth. In order to maintain efficiency, pumping large amounts of geometry through the rendering pipeline would be counterproductive. So instead, we utilize geometry shaders to expand lines into depth-dependent quads, and use pixel shaders (or fragment shaders) to compute the appropriate normals. For distant (sub-pixel) edges, standard line rendering is still used both to increase the

performance, as lines are simpler to compute, and to avoid sub-pixel rasterization issues, such as culling the polygons of the edges entirely.

While a graph visualization should benefit from any number of these techniques, there are critical performance limitations to consider. Current HMD rendering techniques require to render two views separately, one for each eye, which leads to significant performance drop. For a HMD, high frame rate and low latency tracking system are both of critical importance in order to avoid inducing nausea in the user. As such, it is imperative for a visualization to run with frame rate as close to the display's refresh rate limit as possible. The current generation of Oculus Rift, (DK2), runs at 75 Hz, and future HMDs will run at 90 Hz or higher. To get a high frame rate while still attaining good rendering quality, we used Unreal Engine 4 [7] to employ real time rendering technologies which have been extensively optimized by the game industry.

5.2.5 Interaction Techniques

Interfaces within HMD environments is an area of many open challenges and opportunities. Since the user loses direct sight of their hands from within the device, traditional mouse and keyboard interaction is limited. While hand tracking devices such as Leap Motion [8] are gaining popularity in conjunction with virtual reality environments, we found that the currently available generation of the hand tracking devices were not well suited to the tasks of our user study, and users are not as familiar with them as they are with traditional input devices. Thus, we still wanted to use a basic cursor paradigm, as most users are familiar with mouse interaction. In all cases, the 3D selection follows a ray from the center of the view through the cursor.

We considered three methods of controlling a cursor within a HMD environment:

- Cursor follows center of the user's view, without mouse control
- Cursor follows relative to the user's view, with mouse control
- Cursor does not follow the user's view (stay in the world), mouse control only

The first (and simplest) of these options is to lock the cursor to the center of the user's view. Then, the cursor is moved entirely via head tracking. This is the simplest method since it eliminates the need for additional mouse control, other than simple

button interactions. Also, the cursor will always be in the user's view. However, there is no freedom to change the cursor position while maintaining a particular view, and precision requires precise neck motion which can tire the user.

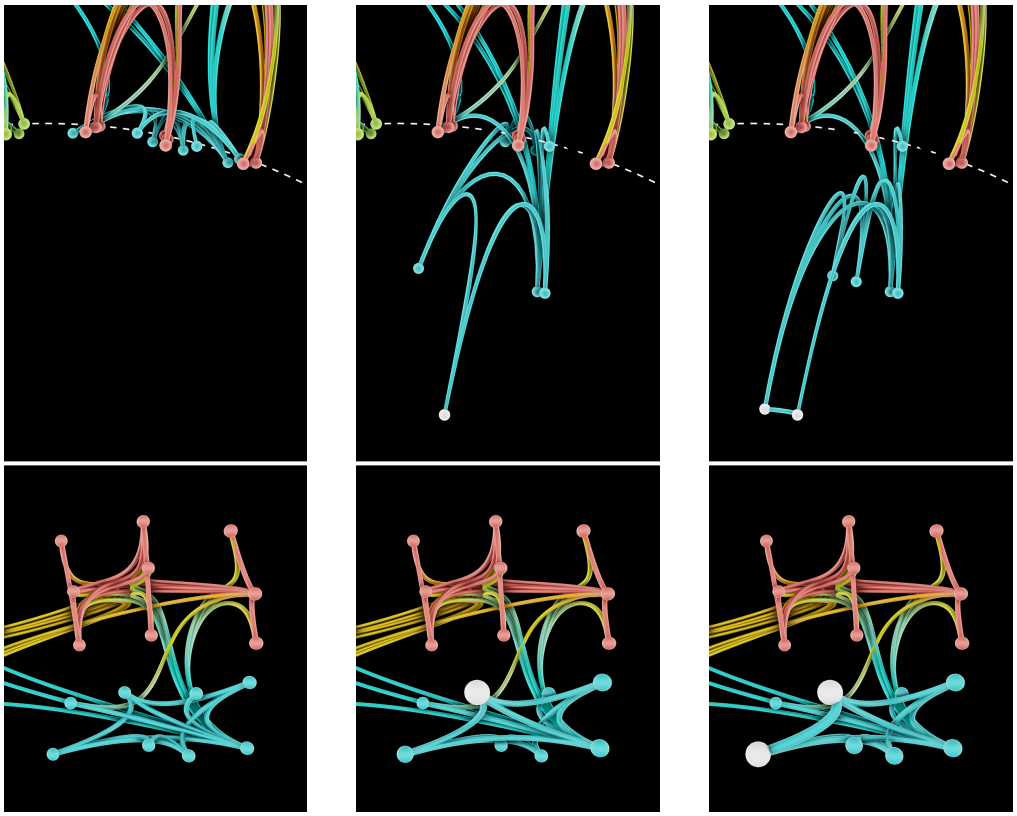
The second method option is to still have the cursor track with the user's view, but to use the mouse to move the cursor within the view. This offers the advantages of keeping the cursor always in the user's view, while allowing the user to fine tune the mouse with a traditional mouse input. However, in our pilot study the users found this method to be confusing. When having the cursor's position depend on both head motion and mouse movement, the cursor to move a lot, often in unintuitive ways. Also, when user changes their view while the cursor is periphery of the view, many unexpected and distracting interactions occur.

The last option is to control the cursor via the mouse relative to the fixed spherical space, regardless of the user's view direction. This is the most similar to the 2D displays that users are used to, because in traditional desktop environments, the cursor keeps its position on the 2D screen space even when user looks at different portions of the screen. However, the downside to this approach is that the cursor can leave or be left outside of the user's view. To counter this, when the cursor is outside of view frustum, we show an arrow that pointing cursor's position, and we added a key shortcut to reset the cursor's position to the center of the user's view. Our pilot study found this approach to be the most effective, so we used this option for our user studies.

5.2.6 Interactive Highlighting

The cursor provides a method of interacting with the graph, but there is also flexibility in how to represent interactions such as inspection, highlight, or selection. We utilize a modified variant of an existing stereoscopic highlighting technique [11] combined with our depth routing techniques and smooth animated transition to better utilize stereoscopic depth cues.

As shown in 5.9a, the edges are routed outside of the sphere before highlighting. When a node is highlighted (5.9b), the highlighted node is moved closer to the user's viewpoint which is the center of the sphere. Also, its adjacent nodes are brought closer to



(a) Before highlighting

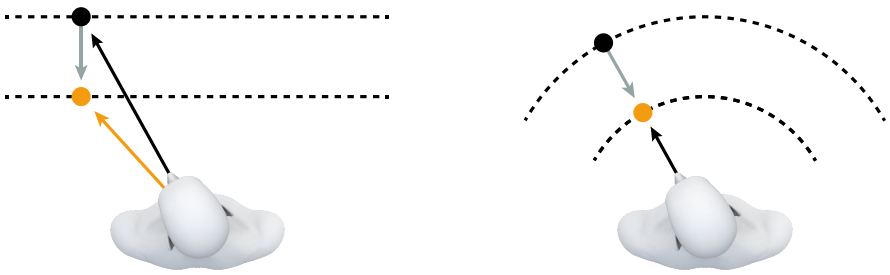
(b) After highlighting a node

(c) Highlighting both nodes in an edge

Figure 5.9. Highlighting technique. The upper row of figures are external views for illustration and the lower row of figures are user's views. (a) Before highlighting, nodes are laid out on the sphere's surface. (b) On highlighting, a node and its neighbors are brought closer to the user, with the highlighted node closer than its neighbors. (c) If two nodes are highlighted, an edge between them is also brought in to the closest focal depth.

the user too, but no more than the highlighted node. The edges that have a highlighted node are also brought closer to differentiate them from other edges. To do this, we transform the depth spline for edge routing of the highlighted edges to be in between the default layout sphere and the focal sphere to bring the edges closer naturally, reducing occlusion or edge crossings for the highlighted edge. The overall effect is that the highlighted nodes/edges are easily distinguished from the remainder of the graph, and the immediate connections are easy to visually follow to their destinations. To further accentuate this, we can apply a halo technique to the highlighted edges.

Multiple nodes/edges can be highlighted and brought into the focal sphere, allowing



(a) 2D layout (b) Spherical layout

Figure 5.10. Stereoscopic highlighting technique [11] on 2D graph layout (a) and spherical graph layout (b).

the user to explore any potentially interesting paths or relations that they find. To emphasize the edges between the highlighted nodes, we remove the depth routing and set the depth to be the same as the highlighted nodes, so that the entire edge spline is on the focal sphere (5.9c). We implemented these depth routing changes with smooth animated transitions for user can keep the track of nodes and edges.

We brought a node closer to the user when they are highlighted it (see Figure 5.10). In spherical layouts (5.10b), the direction of 'getting closer' is same as the direction from the node to the viewpoint. Thus, any other interactions on the highlighted node can be performed without moving the cursor again. In a 2D layout (5.10a), however, if you highlight a node then the position of the node changes in the user's view. So the user needs to move the cursor each time they re-interact with the highlighted node. In fact, the highlighted node in a 2D layout can move to the outside of the user's view.

5.3 User Study

The main purpose of our user study is to evaluate the use of spherical graph layouts and depth routing techniques in immersive graph visualization. The findings from a preliminary pilot study helped set up this study. For example, we found that for many participants, some tasks were either too simple or difficult preventing us from obtaining meaningful results. We also learned that cursor movement for selection should follow mouse movement alone and not depend on head movement. Furthermore, the preliminary study helped find how best to properly label the graph during viewing. These

findings enabled us to remove unwanted factors that would impact the performance of the participants in the new study. The pilot study also helped us narrow the scope of the new study; we dropped the comparison of different displays [179] and focused on the specifics of using a HMD.

5.3.1 Experiment Design

For our study, we designed a within-subjects experiment: 3 *visualization conditions* $\times$ 3 *graph sizes* $\times$ 4 *tasks*. We evaluated three dependent variables in the study: *task completion time*, *correctness rate*, and *number of interactions*. *Task completion time* does not include the time to read the task description. *Correctness rate* is the percentage of tasks correctly answered. *Number of interactions* counts the number of pointing (mouse over on a node), number of highlighting (left click on a node), and number of selecting (right click on a node).

5.3.1.1 Visualization Conditions

We considered three visualization conditions:

- **C1:** 2D graph layout. The graph is laid out on the plane.
- **C2:** Spherical graph layout without depth routing. The graph is laid out on the surface of a sphere.
- **C3:** Spherical graph layout with depth routing. The graph is laid out on the surface of a sphere.

All conditions used our modified stereoscopic highlighting [11], and the same rendering techniques (Section 5.2.4). All participants used a HMD, which is an Oculus Rift DK2. Graph labels always face toward user's viewpoint with the same size.

With the 2D graph layout, we decided not to use depth routing for several reasons. First, depth routed edges are skewed (or narrowed) toward the view direction as a result of perspective distortion. This can be resolved by calculating the depth routing in view space, but if the view changes then the shape of depth routed edges is also changed, which introduces more confusion to the users. Also, the shape of depth routed edges seen by the participant would not be natural if the view direction is not perpendicular to the 2D graph. If we used orthographic projection to render, the participant would

lose the depth perception.

In conditions without depth routing (C1 and C2), we assigned the control points of the 1D depth spline for edge routing (i.e., clusters along a path) by linear interpolation of the depth of the two nodes of the edge.

5.3.1.2 Tasks

We used four tasks in the study:

- **T1:** Find common neighbors. *Select all nodes that are common neighbors of two given nodes, not just one or the other.* The participant can find common neighbors by first highlighting the two given nodes, then look at which nodes have edges connected to the given nodes.
- **T2:** Find the highest degree node. *Among four given nodes, select the node with the highest degree (most neighbors).* User can count the neighbors of each given node by highlighting it.
- **T3:** Find a path. *Find an ordering of a given set of nodes which forms a path along edges from node A to node B, visiting each node once.* The participant was given a start node, an end node, and three other labeled nodes, and instructed that there exists a path that goes through them. The participant was then asked to find the order of the nodes in this path.
- **T4:** Recall node locations. *Find the start node and the end node used in the previous task*. To compare the spatial memory in different visualization conditions, we asked users to remember the locations of the start node and the end node given in T3.

The labels of given nodes are always shown during the tasks, the labels of other nodes are hidden unless they are pointed (mouse over), highlighted (left click), or selected (right click).

5.3.1.3 Graph Size and Field of View

We used three different graphs with different data sizes (i.e., the number of nodes and number of edges) and spatial sizes (FOVs) for the experiment. One additional graph was used for the training session. To keep the visual density of the graph visualizations, we assigned the FOV of a graph according to its size.

It is not necessary to follow the aspect ratio of the HMD for the aspect ratio of the graph layout because the virtual world has infinite space; however, we used 16×9 aspect ratio for all graph layouts according to the estimated range of motion of the participant as discussed in Section 5.2.2.

The graph visualization was located at 10 meter away from the participant's viewpoint in the virtual world. Naturally, the default view direction is the direction from viewpoint to the center of the graph layout. Also, the 2D graph layout (C1) is perpendicular to the default view direction. The distance from the viewpoint to a node is equidistant in a spherical graph layout (C2 and C3), and is closest at the center of a 2D graph layout while the distance increases for nodes away from the center. The geodesic width of a spherical graph layout (C2 and C3) is $r\theta$ and the width of a 2D graph layout (C1) is $2 \cdot tan(\frac{\theta}{2})$, where θ is the horizontal FOV of the graph.

The four datasets are:

D0: This graph [241] consists of 34 nodes and 78 edges. The FOV of this graph is $60° \times 33.75°$. This graph was used in the training session.

D1: The *small* graph [137] consists of 77 nodes and 254 edges. The FOV of this graph is $90° \times 50.625°$.

D2: The *medium* graph [88] consists of 116 nodes and 615 edges. The FOV of this graph is $120° \times 67.5°$.

D3: The *large* graph [227] consists of 297 nodes and 2359 edges. The FOV of this graph is $150° \times 84.375°$.

5.3.2 Participants

We recruited 21 (12 males and 9 females) participants in our user study. The age of the participants ranged from 21 to 34, with the mean age of 25.71 years (SD = 3.96). The group consists of 11 undergraduate students and 10 graduate students. Each participant completed the experiment in about 60 minutes including initial setup, training session, and questionnaires.

Nine participants have normal vision. Ten participants wear eye glasses. Two participants wear contact lenses. All participants are not color-blind. We adjusted HMD for

each participant. The participants wearing glasses used HMD without glasses. However, they had no other vision condition than nearsightedness which can be resolved by adjusting focal distance of HMD. None of the reported differences in vision was statistically significant.

Only one participant had previous experience with HMD. 19 participants had previous experience with stereoscopic viewing from television or movie theater. 14 participants indicated that they had seen a graph visualization (e.g., a node-link diagram) before, and knew what it is, but had never used it. Seven participants had no experience with graph visualization.

5.3.3 Apparatus and Implementation

For a fair comparison of different FOVs for graph visualization, it is necessary to maximize the use of the given area for graph layout. Therefore, the graph layout was calculated using a treemap-based approach [165] in all visualization conditions. To avoid learning effects between tasks, we randomized the graph layout by randomly reordering each level of the hierarchical clustering and randomly orienting intermediate levels of the treemap by multiples of 90°. Nodes were labelled with a three-digit unique random number to remove any possibility of the participants answering the questions from meaningful node labels. The participants used a standard computer mouse as an input device with the spherical cursor technique described in Section 5.2.5.

The participants were seated in front of a desk and used Oculus Rift DK2 [1]. The Oculus Rift DK2 has a 1920×1080 px (split to 960×1080 px per eye) OLED panel with a 75 Hz refresh rate. NVIDIA GTX 980 graphics card was used to render the visualization. The rendering was maintained at about 75 frames per second. The positional tracking was enabled only in C1 because the spherical graph layouts (C2 and C3) do not require the positional tracking.

5.3.4 Procedure

Prior to the beginning of the experiment, we informed the participants of possible issues (e.g., eyestrain, disorientation, or sickness) and right to exit the experiment at any time.

The participants had no time limit for the tasks.

Before each experiment, we adjusted the HMD (for the distance between the pupils of each eye and the distance between HMD lens and the cornea) for each participant. We showed a demo scene about 3 minutes to the participants to identify any issue (e.g., lack of depth perception, eyestrain, disorientation, or sickness) with the HMD. None of the participants had any issue with the HMD.

5.3.4.1 Training

We first instructed the participants about how to use three interaction techniques, pointing (mouse over), highlighting (left click), and selecting (right click). The participants was asked to point, to highlight/dehighlight, to select/deselect specific nodes that the experimenter asked to do so. Each participant first conducted 12 training trials (3 *visualization conditions* $\times$ 4 *tasks*) using D0 to be familiar with the experimental setup and procedure. We then instructed the participants about the strategies for finding a correct answer for each task. During the training, we indicated to the participant whether his/her answer was correct.

5.3.4.2 Experiment

The tasks were presented to the participants in the same repeated order from T1 to T4. The order of *visualization conditions* $\times$ *datasets (graph sizes)* combinations was counterbalanced.

Before each trial, the participant was allowed to rest and could continue to the next trial when ready. Before each trial, we reset the participant's head position and view direction in both physical world and virtual world to remove effect from bad calibrations. The participant read the task description without the graph visualization. After the participant understood the task, then the graph visualization appeared and the task description became hidden to maximize available display area. The task descriptions were shown again when the participant requested it. The participant answered the question by selecting nodes. The participant was asked to talk loud for proceeding to the next step rather than through a GUI to avoid uncontrolled effect from the GUI. During experiment, we did not give any indication to the participant on whether an

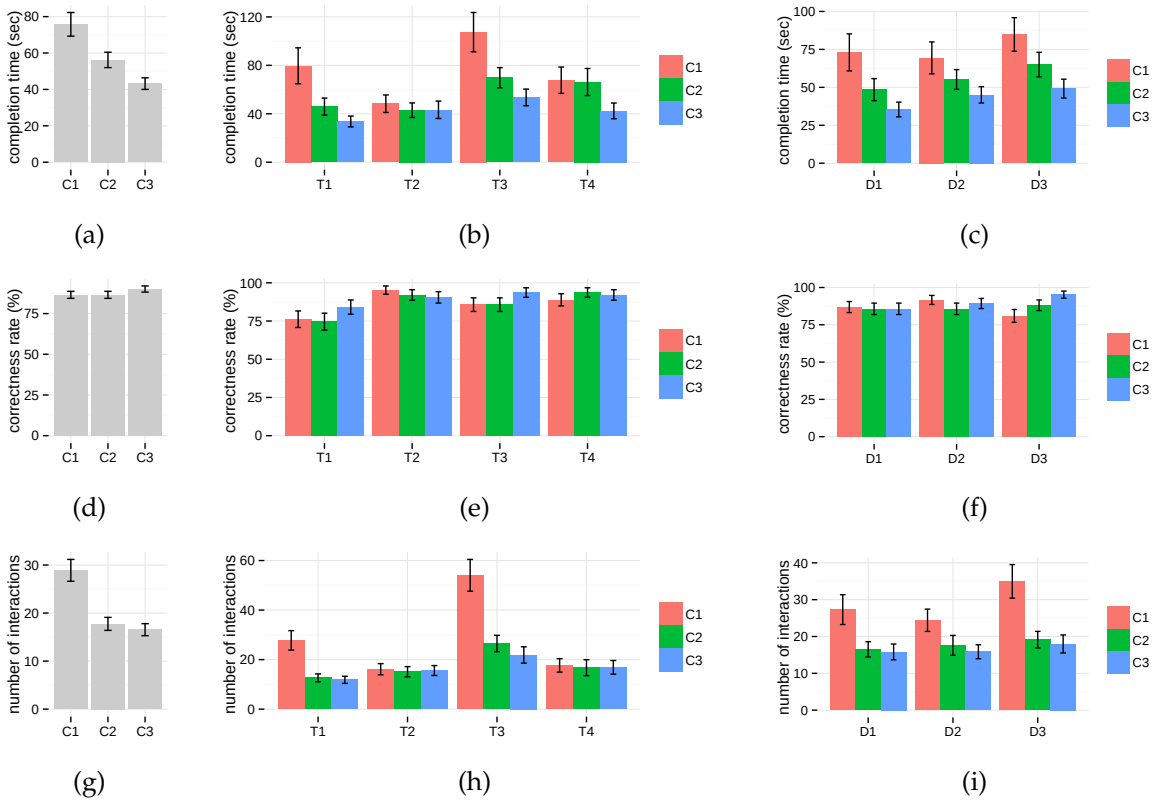


Figure 5.11. Results of the experiment. (a) shows the overall *task completion time* in each *visualization condition*, which is then broken down by *tasks* (b) and *datasets (graph sizes)* (c). Similarly, (d-f) show *correctness rate* and (g-i) show *number of interactions* with the same breakdowns. Overall, using C2 and C3 outperforms C1 in terms of *task completion time* and *number of interactions*. While the overall *correctness rate* is not significantly different between *visualization conditions*, using C3 shows significantly higher *correctness rate* than C1 for the largest graph we tested (D3).

answer was correct or not.

After all trials, the participant was asked to comment on the advantages and disadvantages of the visualization conditions and point out any specific features that he/she liked or disliked in experiment.

5.3.5 Hypotheses

Based on the setting of our user study, we expected to obtain three main results:

H1: For all tasks, spherical graph layouts (C2 and C3) would outperform 2D graph layouts (C1) in immersive environments.

H2: Spherical graph layouts with depth routing (C3) would outperform a spherical

graph layout without depth routing (C2).

5.3.6 Results

Overall, C2 and C3 often outperform C1, and are never outperformed substantially by C1, confirming H1. Similarly, C3 often outperforms C2, as in H2.

5.3.6.1 Task Completion Time

On average, each *task* took 58.39s (SD = 77.86) to complete.

A repeated measures ANOVA with a Greenhouse-Geisser correction (ε = .691) shows a significant effect of *visualization condition* on *task completion time* ($F_{1.38,27.63}$ = 16.05, $p <$ .001). Average *task completion times* (5.11a) are 75.77s for C1 (SD = 103.19), 56.22s for C2 (SD = 67.02), and 43.20s for C3 (SD = 50.55). Post-hoc tests using the Bonferroni correction indicate that C3 is significantly faster than both C1 ($p <$ .0001) and C2 ($p <$ .05), and that C2 is also significantly faster than C1 ($p <$ .05).

We found a significant effect of *task* on *task completion time* ($F_{3,60}$ = 10.23, $p <$ .001). Unsurprisingly, the participants required more time to answer for more complex tasks. Average *task completion times* (5.11b) are 53.06s for T1 (SD = 80.06), 44.86s for T2 (SD = 53.89), 76.89s for T3 (SD = 91.75), and 58.77s for T4 (SD = 77.85).

When we analyzed the results for each *task* separately, there are significant effects of *visualization condition* on *task completion time* for T1 ($F_{1.21,24.15}$ = 6.01, $p <$ .05, ε = 0.604, using Greenhouse-Geisser correction) and T3 ($F_{2,40}$ = 12.68, $p <$ .0001), but not T2 or T4. For T1, both C3 ($p <$ .01) and C2 ($p <$ .05) are significantly faster than C1. For T3, C3 is significantly faster than both C1 ($p <$ .001) and C2 ($p <$ .05), and C2 is also significantly faster than C1 ($p <$ .05).

Our analysis shows a significant effect of *dataset (graph size)* on *task completion time* ($F_{2,40}$ = 3.45, $p <$ .05). As would be expected, the participants required more time to answer for larger graphs. Average *task completion times* (5.11c) are 52.30s for D1 (SD = 80.60), 56.54s for D2 (SD = 71.75), and 66.34s for D3 (SD = 80.54).

5.3.6.2 Correctness Rate

On average, 87.70% of the answers were correct (SD = 17.02) for all 21 participants.

While we did not observe a significant effect of *visualization condition* on *correctness rate* ($F_{2,40} = 2.08$, $p = 0.14$), C3 slightly outperforms the others about 3.5% on average. Average *correctness rate* are 86.51% for C1, 86.51% for C2, and 90.08% for C3 (5.11d). However, there is a significant effect of *visualization condition* on *correctness rate* ($F_{2,40} = 6.92$, $p < .01$) only for the largest graph we tested (D3). For D3, average *correctness rate* are 80.95% for C1, 88.1% for C2, and 95.24% for C3 (5.11f). Post-hoc tests using the Bonferroni correction show C3 is significantly different from C1 ($p < .05$) only for D3.

We found that *correctness rates* significantly differ between *tasks* ($F_{2.09,41.78} = 7.39$, $p < .01$, $\varepsilon = .696$ using Greenhouse-Geisser correction). Unsurprisingly, the participants showed higher *correctness rate* for easier *tasks*. Average *correctness rate* are 78.31% for T1, 92.59% for T2, 88.36% for T3, and 91.53% for T4.

We did not find a significant effect of *dataset (graph size)* on *correctness rate* ($F_{2,40} = 1.23$, $p = .30$). Average *correctness rate* are 88.61% for D1, 88.89% for D2, and 88.1% for D3.

5.3.6.3 Number of Interactions

We measured the *number of interactions* as the sum of number of pointing (mouse over on a node) + number of highlighting (left click on a node) + number of selecting (right click on a node). On average, each *task* took 21.06 interactions ($SD = 27.39$) to complete.

A repeated measures ANOVA with a Greenhouse-Geisser correction ($\varepsilon = .603$) shows a significant effect of *visualization conditions* on *number of interactions* ($F_{1.21,24.13} = 25.63$, $p < .0001$). As shown in 5.11g, average *number of interactions* are 28.89 for C1 ($SD = 36.15$), 17.75 for C2 ($SD = 21.5$), and 16.55 for C3 ($SD = 19.87$). Post-hoc tests using the Bonferroni correction show C1 is significantly different from C2 ($p < .0001$) and C3 ($p < .0001$).

We found a significant effect of *task* on *number of interactions* ($F_{1.58,31.55} = 19.77$, $p < .0001$, $\varepsilon = .526$, using Greenhouse-Geisser correction) As shown in 5.11h, average *number of interactions* are 17.42 for T1 ($SD = 21.65$), 15.6 for T2 ($SD = 16.68$), 34.14 for T3 ($SD = 38.84$), and 17.09 for T4 ($SD = 22.95$).

When we analyzed the results for each *task* separately, there are significant effects

of *visualization condition* on *number of interactions* for T1 ($F_{1.23,24.68} = 14.42$, $p < .001$, $\varepsilon = 0.617$, using Greenhouse-Geisser correction) and T3 ($F_{2,40} = 28.63$, $p < .0001$), but not T2 or T4. For both T1 and T3, both C3 ($p < .001$) and C2 ($p < .001$) show significantly fewer *number of interactions* than C1.

There is a significant effect of *dataset (graph size)* on *number of interactions* ($F_{2,40} = 4.56$, $p < .05$). Average *number of interactions* are 19.87 for D1 ($SD = 27.04$), 19.29 for D2 ($SD = 23.83$), and 24.03 for D3 ($SD = 30.73$).

5.3.6.4 User's Feedback

After the experiment, the participants were asked to answer a post-questionnaire and comment on the experiment freely.

The participants were asked to choose their overall preference between the *visualization conditions* (or *no preference*). All participants except one (who chose *no preference*) preferred C3 over C2 over C1.

All ratings in the questionnaire were measured on a 7 point Likert scale, where toward 1 meant the negative rating (e.g., difficult, not confident), and toward 7 was the positive rating (e.g., easy, confident). We use the Friedman test (non-parametric alternative to one-way repeated measures ANOVA) to test the ratings.

We did not observe a significant difference in confidence ratings (1: no confidence, 7: complete confidence) between the *visualization conditions* ($p = .16$). Mean confidence ratings were 2.95 ($IQR = 2$) for C1, 3.43 ($IQR = 1$) for C2, and 3.62 ($IQR = 2$) for C3.

We found a significant difference in ease-of-use ratings (1: very difficult, 7: very easy) between the *visualization conditions* ($\chi^2(2) = 21.21$, $p < .0001$), with mean ease-of-use ratings on *visualization conditions* were 2.43 ($IQR = 1$) for C1, 4.33 ($IQR = 3$) for C2, and 4.67 ($IQR = 3$) for C3.

The participants responded T3 was the most difficult task and T2 was the easiest task. We found a significant difference in task difficulty ratings (1: very difficult, 7: very easy) between the *tasks* ($\chi^2(3) = 27.68$, $p < .0001$), with mean task difficulty ratings were 3.29 ($IQR = 3$) for T1, 5.38 ($IQR = 2$) for T2, 2.62 ($IQR = 1$) for T3, and 4.52 ($IQR = 2$) for T4. The results of *correctness rate* also indicate that T1 and T3 are more difficult than T2

or T4.

There is a significant difference in ease-of-use ratings (1: very difficult, 7: very easy) between the *datasets (graph sizes)* ($\chi^2(2) = 6.04$, $p < .05$), with mean ease-of-use ratings were 4.14 ($IQR = 3$) for D1, 4.43 ($IQR = 2$) for D2, and 3.33 ($IQR = 3$) for D3.

The participants offered many positive comments for C2 and C3 over C1, such as:

- "It was easier to navigate with C2 and C3",
- "It was hard to control the cursor in C1",
- "The nodes near corner of C1 were too far from me",
- "C1 looks like inclined",
- "When I highlight a node in C1 it looks like new edges are appeared",
- "It was difficult to follow paths in C1".

Six participants asked "can I use zoom feature?" when they are using C1 during the experiment. But no one asks it when they using C2 or C3.

For the depth routing (C3) in particular, some user comments were:

- "It was much less cluttered in C3",
- "I can feel richer depth perception with C3",
- "C1 and C2 look flat",
- "C3 looks like coral, C2 looks like broken egg shells",
- "C3 looks more neat".

Three participants explicitly mentioned about the edge crossings when using the visualization without depth routing (C1 and C2). It is natural because the edges without depth routing are laid on same depth so it cause more edge crossings and depth fighting (z-fighting) problem.

We also asked to the participants to report if they had any kind of discomfort (e.g., eyestrain, disorientation, or sickness). Two participants (one wore contact lenses, another one had normal vision) reported they had light eyestrain, dry eye, and felt pressure on near eye area caused by HMD. No one reported disorientation or sickness. Two participants reported they can clearly see pixel grid of the HMD (also known as the screen door effect) which caused by limited pixel-fill-factor of current HMD. However,

imminent hardware improvements are addressing this in the next generation of HMDs.

5.4 Conclusion

While stereoscopic viewing has previously been shown effective for graph visualization, ours is the first work that achieves immersive graph visualization using a HMD with proven effectiveness. The results of the user study show that participants performed better using our techniques than using traditional 2D graph visualization in an immersive environment, especially for more difficult tasks and larger graphs. This chapter only addresses a few particular design considerations with a limited scope. Even though we have conducted our study with the use of a HMD, our design should also be effective in other immersive virtual reality environments. Our work is an early foray into exploring virtual reality techniques for graph visualization. More research is clearly needed before immersive 3D visualization can begin to benefit real-world information visualization tasks. We hope our work and findings will encourage others to join this exciting area of study so they can help accelerate the development of usable technologies to meet the growing demand of more effective tools for examining and analyzing large complex data.

Chapter 6
Conclusion

Overall, this dissertation research endeavors to answer a fundamental question for visualizing graphs: *How to visualize a graph effectively and efficiently?*

Although existing approaches have mainly focused on developing a faster layout algorithm to compute a node-link diagram, they are still cost-prohibitive for large graphs. On the other hand, the layout results of a graph obtained with expensive computations have only been used for visualizing the input graph itself. Chapter 2 introduces a technique that reuses the existing layouts for visualizing other graphs. As more data becomes available, this approach can provide more accurate previews. With the quick preview, users can make an informed decision on which layout method to use. Thus, this approach can save a tremendous amount of time and resources for visualizing large graphs.

Even if rapid layout methods are available, finding a desired layout for the given circumstance is also a cognitively expensive task due to the vast design space. Much of the previous research on graph visualization has focused on developing an algorithm that computes a "better" layout (i.e., the node positions of a node-link diagram and the node ordering of an adjacency matrix view). However, because of the vast design space, it is nearly impossible to define all the heuristics to find a good layout. In contrast, my approach in Chapter 3 and Chapter 4 is to help users systematically explore diverse visualizations of a graph (node-link diagrams in Chapter 3 and adjacency matrix views

in Chapter 4). Thus, users can effortlessly find the desired visualization that fits the given circumstance. This approach is an example of *artificial intelligence augmentation* [44]: a machine learning model builds a new type of user interface (i.e., layout latent space) to augment a human intelligence task (i.e., graph visualization design).

Emerging immersive technologies introduce new ways of interacting with computers. Consumer-grade immersive technologies (e.g., head-mounted displays) have become ubiquitous that researchers can consider them for general visual analytics. We have found that existing graph visualization techniques are not suitable for immersive environments. Techniques specifically designed for immersive environments are needed. Chapter 5 provides an effective design for visualizing a complex graph in immersive environments.

In conclusion, this dissertation addresses several problems in the field of graph visualization with fundamentally new approaches. I hope my dissertation work will inspire further research into graph visualization.

BIBLIOGRAPHY

[1] Facebook Technologies, LLC. "Oculus Rift." https://www.oculus.com

[2] HTC Corporation. "Vive." http://www.htcvr.com

[3] Sony Computer Entertainment Inc. "PlayStation VR." https://www.playstation.com/en-us/explore/playstation-vr/

[4] Google Inc. "Google Cardboard." https://www.google.com/get/cardboard

[5] Microsoft. "Hololens." https://www.microsoft.com/microsoft-hololens

[6] NASA, "Space Flight Human-System Standard Volumes 1." http://msis.jsc.nasa.gov/volume1.htm

[7] Epic Games. "Unreal Engine 4." https://www.unrealengine.com

[8] Leap Motion, Inc. "Leap Motion." https://www.leapmotion.com

[9] The Supplementary Materials of Chapter 2. http://kwon.io/wgl

[10] The Supplementary Materials of Chapter 3. http://kwon.io/dgl

[11] B. Alper, T. Hollerer, J. Kuchera-Morin, and A. Forbes. Stereoscopic Highlighting: 2D Graph Visualization on Stereo Displays. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2325–2333, 2011.

[12] D. Archambault, T. Munzner, and D. Auber. TopoLayout: Multilevel Graph Layout by Topological Features. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):305–317, 2007.

[13] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer Normalization. *arXiv preprint*, arXiv:1607.06450, 2016.

[14] B. Bach, A. Spritzer, E. Lutton, and J.-D. Fekete. Interactive Random Graph Generation with Evolutionary Algorithms. In *Proc. Graph Drawing*, pages 541–552, 2012.

[15] D. C. Banks and C.-F. Westin. Global Illumination of White Matter Fibers from DT-MRI Data. In *Visualization in Medicine and Life Sciences*, pages 173–184. 2008.

[16] Z. Bar-Joseph, D. K. Gifford, and T. S. Jaakkola. Fast Optimal Leaf Ordering for Hierarchical Clustering. *Bioinformatics*, 17(Suppl. 1):22–29, 2001.

[17] A.-L. Barabási. *Network Science*. Cambridge University Press, 2016.

[18] F. Barahimi and S. Wismath. 3D Graph Visualization with the Oculus Rift. In *Poster Proc. Graph Drawing*, 2014.

[19] H. J. C. Barbosa and A. M. S. Barreto. An Interactive Genetic Algorithm with Co-evolution of Weights for Multiobjective Problems. In *Proc. Annual Conference on Genetic and Evolutionary Computation*, pages 203–210, 2001.

[20] S. T. Barnard, A. Pothen, and H. D. Simon. A Spectral Algorithm for Envelope Reduction of Sparse Matrices. In *Proc. ACM/IEEE Conference on Supercomputing*, pages 493–502, 1993.

[21] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An Open Source Software for Exploring and Manipulating Networks. In *Proc. International AAAI Conference on Weblogs and Social Media*, pages 361–362, 2009.

[22] M. Behrisch, B. Bach, N. Henry Riche, T. Schreck, and J.-D. Fekete. Matrix Reordering Methods for Table and Network Visualization. *Computer Graphics Forum*, 35(3):693–716, 2016.

[23] M. Behrisch, B. Bach, M. Hund, M. Delz, L. V. Rüden, J. D. Fekete, and T. Schreck. Magnostics: Image-Based Search of Interesting Matrix Views for Guided Network Exploration. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):31–40, 2017.

[24] M. Behrisch, T. Schreck, and H. Pfister. GUIRO: User-Guided Matrix Reordering. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):184–194, 2020.

[25] C. Bennett, J. Ryall, L. Spalteholz, and A. Gooch. The Aesthetics of Graph Visualization. In *Proc. Eurographics Conference on Computational Aesthetics in Graphics, Visualization, and Imaging*, Computational Aesthetics, pages 57–64, 2007.

[26] R. Bennett, D. J. Zielinski, and R. Kopper. Comparison of Interactive Environments for the Archaeological Exploration of 3D Landscape Data. In *Proc. IEEE VIS Workshop on 3DVis*, pages 67–71, 2014.

[27] J. Bergstra and Y. Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13(1):281–305, 2012.

[28] J. C. Bezdek and R. J. Hathaway. VAT: A Tool for Visual Assessment of (Cluster) Tendency. In *Proc. International Joint Conference on Neural Networks*, volume 3, pages 2225–2230, 2002.

[29] T. C. Biedl, J. Marks, K. Ryall, and S. Whitesides. Graph Multidrawing: Finding Nice Drawings Without Defining Nice. In *Proc. Graph Drawing*, pages 347–355, 1998.

[30] J. Blythe, C. McGrath, and D. Krackhardt. The Effect of Graph Layout on Inference from Social Network Data. In *Proc. Graph Drawing*, pages 40–51, 1996.

[31] K. M. Borgwardt and H. P. Kriegel. Shortest-Path Kernels on Graphs. In *Proc. IEEE International Conference on Data Mining*, pages 74–81, 2015.

[32] K. M. Borgwardt, T. Petri, S. V. N. Vishwanathan, and H.-P. Kriegel. An Efficient Sampling Scheme for Comparison of Large Graphs. In *Proc. Mining and Learning with Graphs*, 2007.

[33] M. Bostock. Force-Directed Graph Layout Using Velocity Verlet Integration. `https://github.com/d3/d3-force`, 2011.

[34] M. Bostock, V. Ogievetsky, and J. Heer. D^3: Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[35] U. Brandes and C. Pich. Eigensolver Methods for Progressive Multidimensional Scaling of Large Data. In *Proc. Graph Drawing*, pages 42–53, 2006.

[36] R. Brath. 3D InfoVis is Here to Stay: Deal with It. In *Proc. IEEE VIS Workshop on 3DVis*, pages 25–31, 2014.

[37] F. P. Brooks, Jr. What's Real About Virtual Reality? *IEEE Computer Graphics and Applications*, 19(6):16–27, 1999.

[38] M. J. Brusco, H.-F. Köhn, and S. Stahl. Heuristic Implementation of Dynamic Programming for Matrix Permutation Problems in Combinatorial Data Analysis. *Psychometrika*, 73(3):503–522, 2008.

[39] S. Bryson. Virtual Reality in Scientific Visualization. *Communications of the ACM*, 39(5):62–71, 1996.

[40] H. Bunke and G. Allermann. Inexact Graph Matching for Structural Pattern Recognition. *Pattern Recognition Letters*, 1(4):245–253, 1983.

[41] S. R. Buss and J. P. Fillmore. Spherical Averages and Applications to Spherical Splines and Interpolation. *ACM Transactions on Graphics*, 20(2):95–126, 2001.

[42] Y. Cao, J. Xu, S. Lin, F. Wei, and H. Hu. GCNet: Non-local Networks Meet Squeeze-Excitation Networks and Beyond. In *Proc. IEEE/CVF International Conference on Computer Vision Workshop*, pages 1971–1980, 2019.

[43] G. Caraux and S. Pinloche. PermutMatrix: A Graphical Environment to Arrange Gene Expression Profiles in Optimal Linear Order. *Bioinformatics*, 21(7):1280–1281, 2004.

[44] S. Carter and M. Nielsen. Using Artificial Intelligence to Augment Human Intelligence. *Distill*, 2017. https://distill.pub/2017/aia

[45] C.-C. Chang and C.-J. Lin. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27, 2011.

[46] C.-h. Chen. Generalized Association Plots: Information Visualization via Iteratively Generated Correlation Matrices. *Statistica Sinica*, 12:7–29, 2002.

[47] W. Chen, F. Guo, D. Han, J. Pan, X. Nie, J. Xia, and X. Zhang. Structure-Based Suggestive Exploration: A New Approach for Effective Exploration of Large Networks. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):555–565, 2019.

[48] X. Chen, Y. Li, P. Wang, and J. C. Lui. A General Framework for Estimating Graphlet Statistics via Random Walk. *Proc. VLDB Endowment*, 10(3):253–264, 2016.

[49] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, and P. Mutzel. The Open Graph Drawing Framework (OGDF). In R. Tamassia, editor, *Handbook of Graph Drawing and Visualization*, chapter 17. CRC Press, 2013.

[50] A. Cimikowski and P. Shope. A Neural-Network Algorithm for a Graph Layout Problem. *IEEE Transactions on Neural Networks and Learning Systems Information for Authors*, 7(2):341–345, 1996.

[51] A. Civril, M. Magdon-Ismail, and E. Bocek-Rivele. SDE: Graph Drawing Using Spectral Distance Embedding. In *Proc. Graph Drawing*, pages 512–513, 2005.

[52] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). In *Proc. International Conference on Learning Representations*, 2016.

[53] S. Climer and W. Zhang. Rearrangement Clustering: Pitfalls, Remedies, and Applications. *Journal of Machine Learning Research*, 7:919–943, 2006.

[54] J. D. Cohen. Drawing Graphs to Convey Proximity: An Incremental Arrangement Method. *ACM Transactions on Computer-Human Interaction*, 4(3):197–229, 1997.

[55] C. Cortes and V. Vapnik. Support-Vector Networks. *Machine Learning*, 20(3):273–297, 1995.

[56] L. d. F. Costa, F. A. Rodrigues, G. Travieso, and P. R. Villas Boas. Characterization of Complex Networks: A Survey of Measurements. *Advances in Physics*, 56(1):167–242, 2007.

[57] H. S. M. Coxeter. *Introduction to Geometry*. Wiley, 1969.

[58] J. E. Cutting and P. M. Vishton. Perceiving Layout and Knowing Distances: The Integration, Relative Potency, and Contextual Use of Different Information about Depth. In W. Epstein and S. Rogers, editors, *Perception of Space and Motion*, Handbook of Perception and Cognition, pages 69–117. Academic Press, 1995.

[59] H. Dai, Z. Kozareva, B. Dai, A. Smola, and L. Song. Learning Steady-States of Iterative Algorithms over Graphs. In *Proc. International Conference on Machine Learning*, pages 1106–1114, 2018.

[60] R. Davidson and D. Harel. Drawing Graphs Nicely Using Simulated Annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[61] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1, 2011.

[62] C. de Boor. On Calculating with B-Splines. *Journal of Approximation Theory*, 6(1):50–62, 1972.

[63] N. De Cao and T. Kipf. MolGAN: An Implicit Generative Model for Small Molecular Graphs. *Proc. ICML 2018 Workshop on Theoretical Foundations and Applications of Deep Generative Models*, 2018.

[64] M. Dehmer and F. Emmert-Streib, editors. *Quantitative Graph Theory: Mathematical Foundations and Applications*. Chapman and Hall/CRC Press, 2014.

[65] J. M. Dennis. Partitioning with Space-Filling Curves on the Cubed-Sphere. In *Proc. Parallel and Distributed Processing Symposium*, pages 6–11, 2003.

[66] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for Drawing Graphs: An Annotated Bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, 1994.

[67] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.

[68] C. Ding and X. He. Linearized Cluster Assignment via Spectral Ordering. In *Proc. International Conference on Machine Learning*, 2004.

[69] R. dos Santos Vieira, H. A. D. do Nascimento, and W. B. da Silva. The Application of Machine Learning to Problems in Graph Drawing A Literature Review. In *Proc. International Conference on Information, Process, and Knowledge Management*, pages 112–118, 2015.

[70] C. Dunne and B. Shneiderman. Improving Graph Drawing Readability by Incorporating Readability Metrics: A Software Tool for Network Analysts. Technical Report HCIL-2009-13, University of Maryland, 2009.

[71] P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149–160, 1984.

[72] P. Eades, S.-H. Hong, A. Nguyen, and K. Klein. Shape-Based Quality Metrics for Large Graph Visualization. *Journal of Graph Algorithms and Applications*, 21(1):29–53, 2017.

[73] D. Earle and C. B. Hurley. Advances in Dendrogram Seriation for Application to Visualization. *Journal of Computational and Graphical Statistics*, 24(1):1–25, 2015.

[74] S. Eichelbaum, M. Hlawitschka, and G. Scheuermann. LineAO—Improved Three-Dimensional Line Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):433–445, 2013.

[75] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein. Cluster Analysis and Display of Genome-Wide Expression Patterns. *Proc. National Academy of Sciences*, 95(25):14863–14868, 1998.

[76] F. Emmert-Streib, M. Dehmer, and Y. Shi. Fifty Years of Graph Matching, Network Alignment and Network Comparison. *Information Sciences*, 346–347:180–197, 2016.

[77] O. Ersoy, C. Hurter, F. Paulovich, G. Cantareiro, and A. Telea. Skeleton-Based Edge Bundling for Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2364–2373, 2011.

[78] M. H. Everts, H. Bekker, J. B. T. M. Roerdink, and T. Isenberg. Depth-Dependent Halos: Illustrative Rendering of Dense Line Data. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1299–1306, 2009.

[79] M. H. Everts, H. Bekker, J. B. T. M. Roerdink, and T. Isenberg. Flow Visualization Using Illustrative Line Styles. In *Proc. National ICT.Open/SIREN Workshop*, 2011.

[80] A. Febretti, A. Nishimoto, T. Thigpen, J. Talandis, L. Long, J. D. Pirtle, T. Peterka, A. Verlo, M. Brown, D. Plepys, D. Sandin, L. Renambot, A. Johnson, and J. Leigh. CAVE2: A Hybrid Reality Environment for Immersive Simulation and Information Analysis. In *Proc. Engineering Reality of Virtual Reality*, 2013.

[81] A. Frick, A. Ludwig, and H. Mehldau. A Fast Adaptive Layout Algorithm for Undirected Graphs. In *Proc. Graph Drawing*, pages 388–403, 1994.

[82] Y. Frishman and A. Tal. Multi-Level Graph Layout on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1310–1319, 2007.

[83] T. M. J. Fruchterman and E. M. Reingold. Graph Drawing by Force-Directed Placement. *Software: Practice and Experience*, 21(11):1129–1164, 1984.

[84] K. R. Gabriel and R. R. Sokal. A New Statistical Approach to Geographic Variation Analysis. *Systematic Biology*, 18(3):259–278, 1969.

[85] P. Gajer and S. G. Kobourov. GRIP: Graph Drawing with Intelligent Placement. *Journal of Graph Algorithms and Applications*, 6(3):203–224, 2002.

[86] T. Gärtner, P. Flach, and S. Wrobel. On Graph Kernels: Hardness Results and Efficient Alternatives. *Learning Theory and Kernel Machines*, 2777:129–143, 2003.

[87] H. Gibson, J. Faith, and P. Vickers. A Survey of Two-Dimensional Graph Layout Techniques for Information Visualization. *Information Visualization*, 12(3–4):324–357, 2013.

[88] M. Girvan and M. E. J. Newman. Community Structure in Social and Biological Networks. *Proc. National Academy of Sciences*, 99(12):7821–7826, 2002.

[89] P. M. Gleiser and D. Leon. Community Structure in Jazz. *Advances in Complex Systems*, 6(4):565–573, 2003.

[90] R. Gómez-Bombarelli, J. N. Wei, D. Duvenaud, J. M. Hernández-Lobato, B. Sánchez-Lengeling, D. Sheberla, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams, and A. Aspuru-Guzik. Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules. *ACS Central Science*, 4(2):268–276, 2018.

[91] I. Goodfellow. NIPS 2016 Tutorial: Generative Adversarial Networks. *arXiv preprint*, arXiv:1701.00160, 2017.

[92] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Nets. In *Proc. Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

[93] N. Greffard, F. Picarougne, and P. Kuntz. Beyond the Classical Monoscopic 3D in Graph Analytics: An Experimental Study of the Impact of Stereoscopy. In *Proc. IEEE VIS Workshop on 3DVis*, pages 19–24, 2014.

[94] A. Grover, E. Wang, A. Zweig, and S. Ermon. Stochastic Optimization of Sorting Networks via Continuous Relaxations. In *Proc. International Conference on Learning Representations*, 2019.

[95] A. Grover, A. Zweig, and S. Ermon. Graphite: Iterative Generative Modeling of Graphs. In *Proc. International Conference on Machine Learning*, pages 2434–2444, 2019.

[96] G. Gruvaeus and H. Wainer. Two Additions to Hierarchical Cluster Analysis. *British Journal of Mathematical and Statistical Psychology*, 25(2):200–206, 1972.

[97] D. Ha and D. Eck. A Neural Representation of Sketch Drawings. In *Proc. International Conference on Learning Representations*, 2018.

[98] S. Hachul and M. Jünger. Drawing Large Graphs with a Potential-Field-Based Multilevel Algorithm. In *Proc. Graph Drawing*, pages 285–295, 2004.

[99] S. Hachul and M. Jünger. Large-Graph Layout Algorithms at Work: An Experimental Study. *Journal of Graph Algorithms and Applications*, 11(2):345–369, 2007.

[100] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring Network Structure, Dynamics, and Function Using NetworkX. In *Proc. SciPy*, pages 11–15, 2008.

[101] M. Hahsler. An Experimental Comparison of Seriation Methods for One-Mode Two-Way Data. *European Journal of Operational Research*, 257(1):133–143, 2017.

[102] M. Hahsler, K. Hornik, and C. Buchta. Getting Things in Order: An Introduction to the R Package seriation. *Journal of Statistical Software, Articles*, 25(3):1–34, 2008.

[103] H. Haleem, Y. Wang, A. Puri, S. Wadhwa, and H. Qu. Evaluating the Readability of Force Directed Graph Layouts: A Deep Learning Approach. *IEEE Computer Graphics and Applications*, 39(4):40–53, 2019.

[104] H. Halpin, D. J. Zielinski, R. Brady, and G. Kelly. Exploring Semantic Social Networks Using Virtual Reality. In *Proc. International Semantic Web Conference*, pages 599–614, 2008.

[105] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive Representation Learning on Large Graphs. In *Proc. Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[106] D. Harel and Y. Koren. A Fast Multi-Scale Method for Drawing Large Graphs. *Journal of Graph Algorithms and Applications*, 6(3):179–202, 2002.

[107] D. Harel and Y. Koren. Graph Drawing by High-Dimensional Embedding. *Journal of Graph Algorithms and Applications*, 8(2):195–214, 2004.

[108] D. Haussler. Convolution Kernels on Discrete Structures. Technical Report UCSC-CRL-99-10, University of California, Santa Cruz, 1999.

[109] N. Henry and J. Fekete. MatrixExplorer: A Dual-Representation System to Explore Social Networks. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):677–684, 2006.

[110] I. Herman, G. Melançon, and M. S. Marshall. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[111] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner. β-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework. In *Proc. International Conference on Learning Representations*, 2017.

[112] F. Hodson, D. Kendall, and P. Tautu. Seriation from Abundance Matrices. In D. Kendall, editor, *Mathematics in the Archaeological and Historical Sciences*, pages 215–252. Edinburgh University Press, 1971.

[113] D. Holten. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.

[114] D. Holten and J. J. van Wijk. Force-Directed Edge Bundling for Graph Visualization. In *Proc. Eurographics/IEEE-VGTC Conference on Visualization*, pages 983–998, 2009.

[115] T. Horváth, T. Gärtner, and S. Wrobel. Cyclic Pattern Kernels for Predictive Graph Mining. In *Proc. ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 158–167, 2004.

[116] Y. Hu. Efficient and High Quality Force-Directed Graph Drawing. *Mathematica Journal*, 10(1):37–71, 2005.

[117] Z. Hu, Z. Yang, X. Liang, R. Salakhutdinov, and E. P. Xing. Toward Controlled Generation of Text. In *Proc. International Conference on Machine Learning*, pages 1587–1596, 2017.

[118] W. Huang, S.-H. Hong, and P. Eades. Effects of Sociogram Drawing Conventions and Edge Crossings in Social Network Visualization. *Journal of Graph Algorithms and Applications*, 11(2):397–429, 2007.

[119] L. Hubert and J. Schultz. Quadratic Assignment as a General Data Analysis Strategy. *British Journal of Mathematical and Statistical Psychology*, 29(2):190–241, 1976.

[120] T. Hughes, Y. Hyun, and D. Liberles. Visualising Very Large Phylogenetic Trees in Three Dimensional Hyperbolic Space. *BMC Bioinformatics*, 5(1), 2004.

[121] S. Iizuka, E. Simo-Serra, and H. Ishikawa. Let There Be Color!: Joint End-to-End Learning of Global and Local Image Priors for Automatic Image Colorization with Simultaneous Classification. *ACM Transactions on Graphics*, 35(4):110:1–110:11, 2016.

[122] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proc. International Conference on Machine Learning*, pages 448–456, 2015.

[123] M. Jacomy, T. Venturini, S. Heymann, and M. Bastian. ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software. *PLOS ONE*, 9(6):e98679, 2014.

[124] G. F. Jenks. The Data Model Concept in Statistical Mapping. *International Yearbook of Cartography*, 7:186–190, 1967.

[125] B. Jenny. Adaptive Composite Map Projections. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2575–2582, 2012.

[126] F. Kaden. Graphmetriken und Distanzgraphen. *ZKI–Informationen, Akad. Wiss. DDR*, 2(82):1–63, 1982.

[127] T. Kamada and S. Kawai. An Algorithm for Drawing General Undirected Graphs. *Information Processing Letters*, 31(1):7–15, 1988.

[128] T. Karras, T. Aila, S. Laine, and J. Lehtinen. Progressive Growing of GANs for Improved Quality, Stability, and Variation. In *Proc. International Conference on Learning Representations*, 2018.

[129] T. Karras, S. Laine, M. Aittala, J. Hellsten, J. Lehtinen, and T. Aila. Analyzing and Improving the Image Quality of StyleGAN. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, pages 8107–8116, 2020.

[130] H. Kashima, K. Tsuda, and A. Inokuchi. Kernels for Graphs. In K. Tsuda, B. Scholkopf, and J.-P. Vert, editors, *Kernels and Bioinformatics*, pages 155–170. MIT Press, 2004.

[131] J. Katsnelson and S. Kotz. On the Upper Limits of Some Measures of Variability. *Archiv für Meteorologie, Geophysik und Bioklimatologie, Serie B*, 8(1):103–107, 1957.

[132] S. Kieffer, T. Dwyer, K. Marriott, and M. Wybrow. HOLA: Human-like Orthogonal Network Layout. *IEEE Transactions on Visualization and Computer Graphics*, 12(1):349–358, 2016.

[133] M.-J. Kim, M.-S. Kim, and S. Y. Shin. A General Construction Scheme for Unit Quaternion Curves with Simple High Order Derivatives. In *Proc. Annual Conference on Computer Graphics and Interactive Techniques*, pages 369–376, 1995.

[134] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. In *Proc. International Conference on Learning Representations*, 2014.

[135] T. N. Kipf and M. Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proc. International Conference on Learning Representations*, 2016.

[136] M. Klammler, T. Mchedlidze, and A. Pak. Aesthetic Discrimination of Graph Layouts. In *Proc. Graph Drawing and Network Visualization*, pages 169–184, 2018.

[137] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, 1993.

[138] S. G. Kobourov and K. Wampler. Non-euclidean Spring Embedders. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):757–767, 2005.

[139] S. Kolouri, P. E. Pope, C. E. Martin, and G. K. Rohde. Sliced-Wasserstein Auto-Encoders. In *Proc. International Conference on Learning Representations*, 2019.

[140] Y. Koren. Drawing Graphs by Eigenvectors: Theory and Practice. *Computers and Mathematics with Applications*, 49(11–12):1867–1888, 2005.

[141] Y. Koren, L. Carmel, and D. Harel. ACE: A Fast Multiscale Eigenvectors Computation for Drawing Huge Graphs. In *Proc. IEEE Symposium on Information Vissualization*, pages 137–144, 2002.

[142] J. F. Kruiger, P. E. Rauber, R. M. Martins, A. Kerren, S. Kobourov, and A. C. Telea. Graph Layouts by t-SNE. *Computer Graphics Forum*, 36(3):283–294, June 2017.

[143] H. W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, 2(1–2):83–97, 1955.

[144] J. Kunegis. KONECT – The Koblenz Network Collection. In *Proc. International Conference on World Wide Web Companion*, pages 1343–1350, 2013.

[145] O.-H. Kwon, T. Crnovrsanin, and K.-L. Ma. What Would a Graph Look Like in This Layout? A Machine Learning Approach to Large Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):478–488, 2018.

[146] O.-H. Kwon and K.-L. Ma. A Deep Generative Model for Graph Layout. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):665–675, 2020.

[147] O.-H. Kwon, C. Muelder, K. Lee, and K.-L. Ma. Spherical Layout and Rendering Methods for Immersive Graph Visualization. In *Proc. IEEE Pacific Visualization Symposium*, pages 63–67, 2015.

[148] O.-H. Kwon, C. Muelder, K. Lee, and K.-L. Ma. A Study of Layout, Rendering, and Interaction Methods for Immersive Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(7):1802–1815, 2016.

[149] Y. LeCun, Y. Bengio, and G. Hinton. Deep Learning. *Nature*, 521(7553):436–444, 2015.

[150] J. Leskovec and A. Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data`, June 2014.

[151] I. Liiv. Seriation and Matrix Reordering Methods: An Historical Overview. *Statistical Analysis and Data Mining*, 3(2):70–91, 2010.

[152] M. Luboschik and H. Schumann. Illustrative Halos in Information Visualization. In *Proc. Working Conference on Advanced Visual Interfaces*, pages 384–387, 2008.

[153] T. Ma, J. Chen, and C. Xiao. Constrained Generation of Semantically Valid Graphs via Regularizing Variational Autoencoders. In *Proc. Advances in Neural Information Processing Systems*, pages 7113–7124, 2018.

[154] O. Mallo, R. Peikert, C. Sigg, and F. Sadlo. Illuminated Lines Revisited. In *Proc. IEEE Visualization*, pages 19–26, 2005.

[155] T. Masui. Evolutionary Learning of Graph Layout Constraints from Examples. In *Proc. ACM Symposium on User Interface Software and Technology*, pages 103–108, 1994.

[156] J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. In *Proc. Advances in Neural Information Processing Systems*, pages 539–547, 2012.

[157] C. McGrath, J. Blythe, and D. Krackhardt. Seeing Groups in Graph Layout. *Connections*, 19(2):22–29, 1996.

[158] C. McGrath, J. Blythe, and D. Krackhardt. The Effect of Spatial Arrangement on Judgments and Errors in Interpreting Graphs. *Social Networks*, 19(3):223–242, 1997.

[159] J. P. McIntire and K. K. Liggett. The (Possible) Utility of Stereoscopic 3D Displays for Information Visualization: The Good, the Bad, and the Ugly. In *Proc. IEEE VIS Workshop on 3DVis*, pages 1–9, 2014.

[160] F. Mèmoli. Gromov–Wasserstein Distances and the Metric Approach to Object Matching. *Foundations of Computational Mathematics*, 11:417–487, 2011.

[161] G. Mena, D. Belanger, S. Linderman, and J. Snoek. Learning Latent Permutations with Gumbel-Sinkhorn Networks. In *Proc. International Conference on Learning Representations*, 2018.

[162] B. Meyer. Self-Organizing Graphs – A Neural Network Perspective of Graph Layout. In *Proc. Graph Drawing*, pages 246–262, 1998.

[163] T. Milenković, V. Memišević, A. K. Ganesan, and N. Pržulj. Systems-Level Cancer Gene Identification from Protein Interaction Network Topology applied to Melanogenesis-Related Functional Genomics Data. *Journal of the Royal Society Interface*, 7(44):423–437, 2010.

[164] K. Mirhosseini, Q. Sun, K. C. Gurijala, B. Laha, and A. E. Kaufman. Benefits of 3D Immersion for Virtual Colonoscopy. In *Proc. IEEE VIS Workshop on 3DVis*, pages 75–79, 2014.

[165] C. W. Muelder and K.-L. Ma. A Treemap Based Method for Rapid Layout of Large Graphs. In *Proc. IEEE Pacific Visualization Symposium*, pages 231–238, 2008.

[166] C. W. Muelder and K.-L. Ma. Rapid Graph Layout Using Space Filling Curves. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1301–1308, 2008.

[167] T. Munzner. H3: Laying out Large Directed Graphs in 3D Hyperbolic Space. In *Proc. IEEE Symposium on Information Visualization*, pages 2–10, 1997.

[168] T. Munzner. Exploring Large Graphs in 3D Hyperbolic Space. *IEEE Computer Graphics and Applications*, 18(4):18–23, 1998.

[169] T. Munzner. *Visualization Analysis and Design*. CRC Press, 2014.

[170] M. E. J. Newman. Finding Community Structure in Networks Using the Eigenvectors of Matrices. *Physical Review E*, 74:036104, 2006.

[171] Q. H. Nguyen, S. H. Hong, P. Eades, and A. Meidiana. Proxy Graph: Visual Quality Metrics of Big Graph Sampling. *IEEE Transactions on Visualization and Computer Graphics*, 23(6):1600–1611, 2017.

[172] O. Niggemann and B. Stein. A Meta Heuristic for Graph Drawing: Learning the Optimal Graph-Drawing Method for Clustered Graphs. In *Proc. Working Conference on Advanced Visual Interfaces*, pages 286–289, 2000.

[173] Nino Shervashidze. *Scalable Graph Kernels*. PhD thesis, Universität Tübingen, 2012.

[174] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic Differentiation in PyTorch. In *Proc. Advances in Neural Information Processing Systems 2017 Autodiff Workshop*, 2017.

[175] T. P. Peixoto. Hierarchical Block Structures and High-Resolution Model Selection in Large Networks. *Physical Review X*, 4:011047, 2014.

[176] T. P. Peixoto. The graph-tool Python Library. `https://graph-tool.skewed.de`, 2014.

[177] J. Petit. Experiments on the Minimum Linear Arrangement Problem. *ACM Journal of Experimental Algorithmics*, 8:2.3, 2004.

[178] G. Peyré, M. Cuturi, and J. Solomon. Gromov-Wasserstein Averaging of Kernel and Distance Matrices. In *Proc. International Conference on Machine Learning*, pages 2664–2672, 2016.

[179] Prabhat, A. Forsberg, M. Katzourin, K. Wharton, and M. Slater. A Comparative Study of Desktop, Fishtank, and Cave Systems for the Exploration of Volume Rendered Confocal Data Sets. *IEEE Transactions on Visualization and Computer Graphics*, 14(3):551–563, 2008.

[180] S. Prillo and J. M. Eisenschlos. SoftSort: A Continuous Relaxation for the argsort Operator. In *Proc. International Conference on Machine Learning*, pages 7793–7802, 2020.

[181] N. Przulj. Biological Network Comparison Using Graphlet Degree Distribution. *Bioinformatics*, 23(2):e177–e188, 2007.

[182] N. Przulj, D. G. Corneil, and I. Jurisica. Modeling Interactome: Scale-Free or Geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[183] H. C. Purchase. Metrics for Graph Drawing Aesthetics. *Journal of Visual Languages and Computing*, 13(5):501–516, 2002.

[184] H. C. Purchase, J.-A. Allder, and D. Carrington. Graph Layout Aesthetics in UML Diagrams: User Preferences. *Journal of Graph Algorithms and Applications*, 6(3):255–279, 2002.

[185] H. C. Purchase, C. Pilcher, and B. Plimmer. Graph Drawing Aesthetics—Created by Users, Not Algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 18(1):81–92, 2012.

[186] A. Radford, L. Metz, and S. Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *arXiv preprint*, arXiv:1511.06434, 2015.

[187] M. Rahman, M. A. Bhuiyan, M. Rahman, and M. A. Hasan. GUISE: A Uniform Sampler for Constructing Frequency Histogram of Graphlets. *Knowledge and Information Systems*, 38(3):511–536, 2013.

[188] P. Ramachandran, N. Parmar, A. Vaswani, I. Bello, A. Levskaya, and J. Shlens. Stand-Alone Self-Attention in Vision Models. In *Advances in Neural Information Processing Systems*, pages 68–80. 2019.

[189] J. Ramon and T. Gärtner. Expressivity versus Efficiency of Graph Kernels. In *Proc. International Workshop on Mining Graphs, Trees and Sequences*, 2003.

[190] S. J. Reddi, S. Kale, and S. Kumar. On the Convergence of Adam and Beyond. In *Proc. International Conference on Learning Representations*, 2018.

[191] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proc. LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, 2010.

[192] H. Robbins and S. Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[193] A. Roberts, J. Engel, C. Raffel, C. Hawthorne, and D. Eck. A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music. In *Proc. International Conference on Machine Learning*, pages 4364–4373, 2018.

[194] J. W. Scannell, G. A. Burns, C. C. Hilgetag, M. A. O'Neil, and M. P. Young. The Connectional Organization of the Cortico-Thalamic System of the Cat. *Cerebral Cortex*, 9(3):277–299, 1999.

[195] Z. Shen, M. Zhang, S. Yi, J. Yan, and H. Zhao. Efficient Attention: Attention with Linear Complexities. *arXiv preprint*, arXiv:1812.01243, 2018.

[196] N. Shervashidze and K. M. Borgwardt. Fast Subtree Kernels on Graphs. In *Proc. Advances in Neural Information Processing Systems*, pages 1660–1668, 2009.

[197] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

[198] N. Shervashidze, S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt. Efficient Graphlet Kernels for Large Graph Comparison. In *Proc. International Conference on Artificial Intelligence and Statistics*, pages 488–495, 2009.

[199] J. Shi and J. Malik. Normalized Cuts and Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[200] K. Shoemake. Animating Rotation with Quaternion Curves. *ACM SIGGRAPH Computer Graphics*, 19(3):245–254, 1985.

[201] M. Simonovsky and N. Komodakis. GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders. In *Proc. International Conference on Artificial Neural Networks*, pages 412–422, 2018.

[202] A. J. Smola and B. Schölkopf. A Tutorial on Support Vector Regression. *Statistics and Computing*, 14(3):199–222, 2004.

[203] F. Sobik. Graphmetriken und Klassifikation Strukturierter Objekte. *ZKI–Informationen, Akad. Wiss. DDR*, 2(82):63–122, 1982.

[204] M. Spönemann, B. Duderstadt, and R. von Hanxleden. Evolutionary Meta Layout of Graphs. In *Proc. Diagrams*, pages 16–30, 2014.

[205] T. Sprenger, M. Gross, A. Eggenberger, and M. Kaufmann. A Framework for Physically-Based Information Visualization. In *Visualization in Scientific Computing*, pages 71–83. 1997.

[206] R. Tamassia, editor. *Handbook of Graph Drawing and Visualization*. CRC Press, 2013.

[207] M. Tennekes and E. de Jonge. Tree Colors: Color Schemes for Tree-Structured Data. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2072–2081, 2014.

[208] A. Teyseyre and M. Campo. An Overview of 3D Software Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):87–105, 2009.

[209] V. A. Traag, L. Waltman, and N. J. van Eck. From Louvain to Leiden: Guaranteeing Well-Connected Communities. *Scientific Reports*, 9(1):5233, 2019.

[210] D. Tsafrir, I. Tsafrir, L. Ein-Dor, O. Zuk, D. Notterman, and E. Domany. Sorting Points into Neighborhoods (SPIN): Data Analysis and Visualization by Ordering Distance Matrices. *Bioinformatics*, 21(10):2301–2308, 2005.

[211] J. Ugander, L. Backstrom, and J. Kleinberg. Subgraph Frequencies: Mapping the Empirical and Extremal Geography of Large Graph Collections. In *Proc. WWW*, pages 1307–1318, 2013.

[212] R. B. J. van Brakel, M. W. E. J. Fiers, C. Francke, M. A. Westenberg, and H. van de Wetering. COMBat: Visualizing Co-occurrence of Annotation Terms. In *Proc. IEEE Symposium on Biological Data Visualization*, pages 17–24, 2013.

[213] A. van Dam, A. Forsberg, D. Laidlaw, J. LaViola, and R. Simpson. Immersive VR for Scientific Visualization: A Progress Report. *IEEE Computer Graphics and Applications*, 20(6):26–52, 2000.

[214] L. van der Maaten and G. Hinton. Visualizing High-Dimensional Data Using t-SNE. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[215] F. van Ham and B. Rogowitz. Perceptual Organization in User-Generated Graph Layouts. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1333–1339, 2008.

[216] C. Villani. *Topics in Optimal Transportation*, volume 58 of *Graduate Studies in Mathematics*. American Mathematical Society, 2003.

[217] S. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph Kernels. *Journal of Machine Learning Research*, 11:1201–1242, 2010.

[218] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. van Wijk, J.-D. Fekete, and D. Fellner. Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges. *Computer Graphics Forum*, 30(6):1719–1749, 2011.

[219] C. Walshaw. A Multilevel Algorithm for Force-Directed Graph-Drawing. *Journal of Graph Algorithms and Applications*, 7(3):253–285, 2003.

[220] P. Wang, J. C. S. Lui, B. Ribeiro, D. Towsley, J. Zhao, and X. Guan. Efficiently Estimating Motif Statistics of Large Networks. *ACM Transactions on Knowledge Discovery from Data*, 9(2):8, 2014.

[221] R.-L. Wang and K. Okazaki. Artificial Neural Network for Minimum Crossing Number Problem. In *Proc. International Conference on Machine Learning and Cybernetics*, pages 4201–4204, 2005.

[222] X. Wang, R. Girshick, A. Gupta, and K. He. Non-local Neural Networks. In *Prox. IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7794–7803, 2018.

[223] C. Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann, 2004.

[224] C. Ware and G. Franck. Evaluating Stereo and Motion Cues for Visualizing Information Nets in Three Dimensions. *ACM Transactions on Graphics*, 15(2):121–140, 1996.

[225] C. Ware and P. Mitchell. Reevaluating Stereo and Motion Cues for Visualizing Graphs in Three Dimensions. In *Proc. Symposium on Applied Perception in Graphics and Visualization*, pages 51–58, 2005.

[226] C. Ware and P. Mitchell. Visualizing Graphs in Three Dimensions. *ACM Transactions on Applied Perception*, 5(1), 2008.

[227] D. J. Watts and S. H. Strogatz. Collective Dynamics of 'Small-World' Networks. *Nature*, 393:440–442, 1998.

[228] M. Wertheimer. Untersuchungen zur Lehre von der Gestalt. II. *Psychologische Forschung*, 4(1):301–350, 1923.

[229] C. H. B. Weyers, B. Hentschel, and T. W. Kuhlen. Interactive Volume Rendering for Immersive Virtual Environments. In *Proc. IEEE VIS Workshop on 3DVis*, pages 73–74, 2014.

[230] E. Wilkinson. Archaeological Seriation and the Travelling Salesman Problem. *Mathematics in the Archaeological and Historical Sciences*, pages 276–284, 1971.

[231] H.-M. Wu, S. Tzeng, and C.-h. Chen. Matrix Visualization. In C.-h. Chen, W. Härdle, and A. Unwin, editors, *Handbook of Data Visualization*, pages 681–708. Springer, 2008.

[232] Y. Wu, N. Cao, D. Archambault, Q. Shen, H. Qu, and W. Cui. Evaluation of Graph Sampling: A Visualization Perspective. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):401–410, 2017.

[233] Y. Wu and M. Takatsuka. Visualizing Multivariate Network on the Surface of a Sphere. In *Proc. Asia-Pacific Symposium on Information Visualisation*, pages 77–83, 2006.

[234] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How Powerful are Graph Neural Networks? In *Proc. International Conference on Learning Representations*, 2019.

[235] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka. Representation Learning on Graphs with Jumping Knowledge Networks. In *Proc. International Conference on Machine Learning*, pages 5453–5462, 2018.

[236] P. Yanardag and S. Vishwanathan. Deep Graph Kernels. In *Proc. ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1365–1374, 2015.

[237] R. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec. Hierarchical Graph Representation Learning with Differentiable Pooling. In *Proc. Advances in Neural Information Processing Systems*, pages 4800–4810, 2018.

[238] J. You, R. Ying, X. Ren, W. Hamilton, and J. Leskovec. GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models. In *Proc. International Conference on Machine Learning*, pages 5708–5717, 2018.

[239] J. W. Youdas, T. R. Garrett, V. J. Suman, C. L. Bogard, H. O. Hallman, and J. R. Carey. Normal Range of Motion of the Cervical Spine: An Initial Goniometric Study. *Physical Therapy*, 72(11):770–780, 1992.

[240] M. P. Young. The Organization of Neural Systems in the Primate Cerebral Cortex. *Proc. Royal Society of London B: Biological Sciences*, 252(1333):13–18, 1993.

[241] W. W. Zachary. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research*, 30:452–473, 1977.

[242] B. Zelinka. On a Certain Distance between Isomorphism Classes of Graphs. *Časopis pro pěstování matematiky*, 100(4):371–373, 1975.

[243] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena. Self-Attention Generative Adversarial Networks. In *Proc. International Conference on Machine Learning*, pages 7354–7363, 2019.

[244] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang. The Unreasonable Effectiveness of Deep Features as a Perceptual Metric. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, pages 586–595, 2018.