Efficient Embedding of Scale-Free Graphs in a Weighted Geometric Space

Bachelor's Thesis of

Markus Wünstel

At the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	TT-Prof. Dr. Thomas Bläsius
Second reviewer:	Prof. Dr. Dorothea Wagner
Advisors:	Dr. Maximilian Katzmann
	Jean-Pierre von der Heydt

25.04.2023 – 25.08.2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text. I also declare that I have read and observed the *Satzung zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie.*

Karlsruhe, 25.08.2023

Markus Wünstel
. . . .
(Markus Wünstel)

Abstract

Networks in the real world are of strong economical, technical and social importance. A lot of tasks such as link prediction or node classification can be solved on those networks if we have an embedding for the graph of this network that represents its graph structure. One way to obtain an embedding is to assume that the existence of edges in a graph is tied to distances between geometrical representations of the vertices in some hidden geometry. This connection between graph topology and underlying geometry is then formalized using a graph model. A graph model that captures important properties of real world graphs such as a heterogenous degree distribution and a high clustering coefficient is the Geometric Inhomogeneous Random Graph model or the Hyperbolic Random Graph model, which is similar to the previous one.
We develop the first algorithm that embeds a graph according to the Geometric Inhomogeneous Random Graph model. The main advantage over existing embedders is the use of a weighted geometric space as ground space, which is easy to handle and which can have higher dimensions. Our algorithm is a maximum likelihood embedder that means we find parameters for each vertex in a given graph such that the probability to obtain the input graph under a given graph model is maximized. Our approach is based on an existing maximum likelihood embedder for the Hyperbolic Random Graph model. We implement the algorithm and evaluate its performance by measuring the quality of the different parts of the algorithm and the influence of different graph properties on the quality of the embedding and found that our algorithm works quite well.

Zusammenfassung

In der realen Welt sind Netzwerke von großer ökonomischer, technologischer und sozialer Bedeutung. Viele Probleme wie link prediction oder node classification können auf diesen Netzwerken effizient gelöst werden, wenn es eine Einbettung zu diesem Netzwerk gibt, die dessen Graphstruktur repräsentiert. Um eine Einbettung zu erhalten, nehmen wir an, dass die Existenz einer Kante an die Distanz zwischen der geometrischen Repräsentation der Knoten in einer versteckten Geometrie geknüpft ist. Die Verbindung zwischen Graphtopologie und zugrundeliegender Geometrie wird durch Graphmodelle formalisiert. Ein Graphmodell, das wichtige Eigenschaften von realen Netzwerken abbildet, wie eine heterogene Knotengradverteilung oder einen hohen Clusterbildungskoeffizient, ist das Geometric Inhomogeneous Random Graph-Modell oder das Hyperbolic Random Graph-Modell, welches dem vorherigen sehr ähnlich ist.
Wir haben den ersten Algorithmus entwickelt, der Graphen gemäß dem Geometric Inhomogeneous Random Graph-Modell einbettet. Der entscheidende Vorteil zu bereits existierenden Einbettungsalgorithmen ist, dass als Grundraum ein gewichteter geometrischer Raum genutzt wird, der einfacher handzuhaben ist und höhere Dimensionen unterstüzt. Unser Algorithmus ist ein maximum likelihood Einbettungsalgorithmus, dabei versuchen wir die Parameter der Knoten so zu wählen, dass die Wahrscheinlichkeit, den gegebenen Graphen aus dem Graphmodell zu generieren, maximiert wird. Unser Ansatz basiert auf einem bereits bestehenden maximum likelihood Einbettungsalgorithmus für das Hyperbolic Random Graph-Modell. Wir haben unseren Algorithmus implementiert und dessen Qualität evaluiert, indem wir die Qualität der einzelnen Bestandteile des Algorithmus und den Einfluss verschiedener Grapheigenschaften auf die Qualität des Algorithmus untersucht haben. Dabei haben wir herausgefunden, dass unser Algorithmus recht gut funktioniert.

Contents

1 Introduction

A lot of networks in the real world, e.g., social networks like Instagram, can be modeled as graphs, consisting of edges and vertices. There are a lot of tasks that need to be performed on those networks. A social network often makes recommendations to follow other people on this network. On a graph this task is called link prediction and is commonly used on networks [LK07]. Another task could be the identification of communities in a network. In a social network an example for this is the identification of friendship groups. Sorting a group of vertices in the same category, e.g., products in the Amazon recommendation network, is called node classification. Other task for networks are visualization, routing or the spread of epidemics [BS03].
The link prediction and clustering problems can be tackled if we introduce a measure of closeness. To solve the link prediction problem, we can predict a new edge between vertices if two vertices are close but not yet connected. Communities can be identified by a group of vertices that are close together. A way to measure closeness in a network is the concept of *embeddings*. Embeddings enable us to model real world properties of graphs. In an embedding we assign each vertex a position in a metric space. With the distances calculated in the metric space we can measure the closeness between vertices.
There already exist some embedders that embed networks in a metric space. In a previous paper [GF18] a lot of embedding methods are collected, which also aim to solve the previous defined problems such as link prediction, clustering, node classification and visualization. However, the embedding methods always use a d-dimensional Euclidean space. The problem with this is that a lot of real world graphs have a heterogenous degree distribution and cannot be represented well in the Euclidean space because networks with a heterogenous degree distribution have an underlying hyperbolic geometry [Kri+10]. An example for this is the star graph. We want the middle vertex to be near all other vertices while each outside vertex should be near the middle and far away from the other vertices. If there are a lot more vertices than dimensions it is not possible to find such an embedding in the Euclidean Space.
Another category of embedders use the *hyperbolic plane* as metric space for the embedding. The reason for this is that the previously mentioned problem, the heterogeneity, is solved there. A common approach in the Euclidean space are spring embedders. This approach was adapted for the hyperbolic plane [BFK21]. There also exist other maximum likelihood estimation embedders. An example is the *HyperMap* algorithm which iteratively adds vertices and tries to improve the overall likelihood [PPK15]. Another algorithm that produces a maximum likelihood estimation embedding is proposed in [BFKL16]. Here the advantages of a spring embedder and maximum likelihood estimation embedder are combined. This embedder produces good embeddings for the hyperbolic plane.
A problem with the hyperbolic embedders is that they all embed the graph in the (2-dimensional) hyperbolic plane. The assumption there is that all graphs that occur in the real world only have two dimensions which is just unreasonable. This restriction exists because it is difficult to extend this approach to higher dimensions, e.g., for the 3-dimensional Hyperbolic Space.
Another field of study where graph embeddings are used is machine learning. A common approach are *graph neural networks*. They also use the hyperbolic space for embedding their

networks [WHWW21]. In machine learning, knowledge graph embedding (KGE) is used a lot. The Large-scale Information Network Embedding (LINE) [Tan+15] follows the same objective. They typically embed graphs into a high dimensional space, e.g., with 128 dimensions.
The existing embedders have the problem that they embed graphs in either too many or to few dimensions or in the wrong geometry or they do not scale well for large graph.
As mentioned, real world networks can be characterized by various properties such as a *power-law* degree distribution and a *high clustering coefficient* [VHHK19|New01]. These networks are so called *scale-free* networks. We are especially interested in this type of graphs because a lot of real world networks seem to have similar properties.
What is a suitable mental picture for these properties in a network? Again, we can consider the social network Instagram as an example for a scale-free graph. We notice that there are very few users like Cristiano Ronaldo or Selena Gomez who have a lot of followers (a high vertex degree) while the majority of users only have very few followers. This is called a *heterogenous* degree distribution which we model as power-law degree distribution. A high clustering coefficient can be observed with friendship groups. If one has two friends it is very likely that they also follow each other. Those triangles are the result of a high clustering coefficient. It is difficult for Euclidean embedders to find an embedding for a scale-free network.
Fortunately, there are mathematical graph models that generate graphs with these properties such as the Hyperbolic Random Graph (HRG) model which is already well discussed in literature [Kri+10]. We focus on a further model, the Geometric Inhomogeneous Random Graph model [Keu18]. This model has been observed to model real world graphs well [BF22]. The GIRG model uses a *weighted geometric space* as ground space. The degree of a vertex is proportional to its weight and hence the power-law degree distribution is achieved with a power-law distribution of the weights. The high clustering coefficient is achieved through the geometric space. The probability for an edge to exist between two vertices is based on their distance and their weights. We want to find positions for the vertices such that the probability that the graph is generated under a given model like the GIRG model is maximized. An embedder with this goal is called *maximum likelihood embedder*.
We solve the problems of the previous embedders, the heterogenous degree distribution and the fixed dimension, by embedding networks in a weighted geometric space. The advantage is that this space is a lot easier to handle than the hyperbolic space because we can use Euclidean geometry while it also allows us to model the heterogenous degree distribution with the weights. If we just use a 1-dimensional space this model is similar to the HRG model but in contrast to the hyperbolic plane it can be easily extended to higher dimensions. The embedding algorithm we introduce in this thesis is based on an embedder for the hyperbolic plane [BFKL16]. The algorithm consists of two main steps where the first one is to embed the core of the graph with a spring embedder and in the second step we embed the remaining vertices by maximizing the likelihood. Because of the similarity of both spaces, the hyperbolic plane and the 1-dimensional weighted geometric space, we can take a lot of insights about the embedding process from the HRG embedder and apply it to the weighted geometric space. Because the GIRG model is a more generalized model it is more powerful but there are also challenges in the embedding process that are addressed and solved in this work. After the implementation of our newly developed algorithm, we also provide an extensive empirical evaluation of our algorithm.

Outline First we introduce all necessary definitions and notions in Chapter 2. This includes the two graph models, the Hyperbolic Random Graph model and the Geometric Inhomogeneous Random Graph model. We also define some mathematical constructs we use in the thesis, e.g., the log-likelihood. Then we explain in Chapter 3 how the algorithm works that we use as starting point. In this chapter we also explain our new algorithm and the theoretical foundations for that. We perform the necessary calculations adapting it to the GIRG model and show the differences to the HRG model. Then we evaluate our algorithm empirically in Chapter 4. Finally we give a summary and a short outlook for future work on the algorithm in Chapter 5.

2 Preliminaries

In this chapter we introduce all notions and concepts that we will use throughout this thesis. This includes the two geometric random graph models that we use as well as a common embedding technique and the way we measure the quality of embeddings.

2.1 Hyperbolic Random Graphs

The Hyperbolic Random Graph (HRG) model was first introduced in [Kri+10].

2.1.1 Hyperbolic Space

A point p in the hyperbolic plane is represented by radial coordinates (r_p, φ_p). The first parameter r_p describes the hyperbolic distance to the origin. The second parameter φ_p is the angle with respect to the x-axis. The angle is in the interval $[0, 2\pi]$. The distance between two points x and y in the hyperbolic plane is defined as

$$\mathrm{dist}(p, q) := \cosh^{-1}(\cosh(r_p)\cosh(r_q) - \sinh(r_p)\sinh(r_q)\cos(\varphi_p - \varphi_q)).$$

2.1.2 Hyperbolic Random Graphs

In the HRG model a graph is drawn on a disk in the hyperbolic plane. The vertices are distributed (quasi) uniformly at random in this disk. The probability for two vertices to be adjacent depends on the hyperbolic distance between them. The closer they are, the likelier it is that an edge between them exists. The radial coordinate can be seen as a measure of the *popularity* of the vertex. A vertex that is close to the origin is connected to a lot of vertices and a vertex on the border of the disk is only connected to very few vertices. The angular coordinate describes the *similarity* of the vertices. In a real world example like the Amazon product recommendation network, one could imagine that products from the same category have similar angular coordinates and products close to the origin are products that are bought a lot with other products. In [BFKL16] an embedding of the Amazon product recommendation network is shown. Nodes that belong to the same category are placed near (close angular coordinate) such that the categories could be obtained from the embedding although the algorithm did not know the ground truth communities. The HRG depends on four parameters n, c, α and T which are described in the following.

Vertices A graph in the HRG model consists of a set of n vertices $V = \{v_1, \ldots, v_n\}$.

Degree Distribution (α, c) The vertices are distributed on a disk in the hyperbolic plane. This disk has a radius R that can be computed from the parameters n and c as $R = 2\log(n) + c$. The constant $c \in \mathbb{R}^+$ in this equation determines the average degree of the graph and depends on the constant $\alpha \in \left(\frac{1}{2}, 1\right)$. This yields a power-law degree distribution with exponent $\beta = 2\alpha + 1$.

Temperature T The temperature $T \in [0,1]$ controls the clustering. The model for $T > 0$ is called *binomial model*. If $T = 0$ the model is called *threshold model*.
From these parameters we can sample a HRG as follows. The position of a vertex $p_v = (r_v, \varphi_v)$ is drawn randomly in the disk in the hyperbolic plane. The density function of the probability distribution for the angle is

$$f(\varphi_v) = \frac{1}{2\pi}.$$

The radial coordinate is sampled from the probability distribution with density function

$$f(r_v) = \frac{\alpha \sinh(\alpha r_v)}{\cosh(\alpha R) - 1}.$$

Hence the density function of the joint distribution is

$$f(r_v, \varphi_v) = \frac{\alpha \sinh(\alpha r_v)}{2\pi(\cosh(\alpha R) - 1)}.$$

The probability of two vertices u and v being connected in the binomial model is given by

$$p_{uv} = p(\operatorname{dist}(u,v)) = \frac{1}{(1 + e^{\frac{1}{2T}\cdot(\operatorname{dist}(u,v)-R)})}.$$

In the threshold model where $T = 0$ two vertices are connected if and only if their distance is below the radius R, that is

$$p_{uv} = \begin{cases} 1 & \text{if } \operatorname{dist}(u,v) < R \\ 0 & \text{else.} \end{cases}$$

2.2 Geometric Inhomogeneous Random Graphs

A graph in the Geometric Inhomogeneous Random Graph model (GIRG) is drawn on a weighted geometric space [Keu18]. Additionally to its position on a d-dimensional ground space each vertex has a weight. This weight acts like the radial coordinate in the HRG model and has a big influence on the connection probability of two vertices. A vertex with a high weight has a high popularity since it is connected to more vertices than a low weight vertex. The coordinate in the ground space describes the similarity of vertices. But in contrast to the HRG model the ground space can have an arbitrary dimension. The HRG model can be seen as a special case of the 1-dimensional GIRG model. This relation is further explained in Section 2.2.3.

2.2.1 Weighted Geometric Space

A vertex v in the weighted geometric space has a *weight* $w_v \in \mathbb{R}^+$ and a *position* $x_v \in \mathbb{X}^d$ for $\mathbb{X} \in \{\mathbb{T}, [0,1]\}$. We use two different d-dimensional ground spaces for this, a torus which is a unit cube where each two opposite sides are identified or a unit cube without the wrap-around. The removal of the wrap-around of the torus has some advantages and disadvantages. Some calculations that involve distance calculations are easier without the wrap-around whereas other calculations get more complicated because the space around a vertex depends on its position. We use both ground spaces in this thesis. The distance between two vertices u and v on the torus can be measured with the maximum norm L_∞, where

$$\|x_v - x_u\| := \|x_v - x_u\|_\infty = \max_{1 \le i \le d} \min\{|x_{vi} - x_{ui}|, 1 - |x_{vi} - x_{ui}|\}.$$

For the cube the L_∞-norm is defined as

$$\|x_v - x_v\|_\infty = \max_{1 \le i \le d} |x_{vi} - x_{ui}|.$$

In this thesis we mainly focus on the L_∞-norm but we also consider other norms too, for example the Euclidean L_2-norm, where the distance on the torus is

$$\|x_u - x_v\|_2 = \sqrt{\sum_{i=1}^{d} \left(\min\left\{|x_{vi} - x_{ui}|, 1 - |x_{vi} - x_{ui}|\right\}\right)^2}.$$

For the cube the L_2-norm is

$$\|x_v - x_u\|_2 = \sqrt{\sum_{i=1}^{d} (x_{vi} - x_{ui})^2}.$$

2.2.2 Model

In addition to the four parameters n, β, c and T, GIRGs also feature the dimension d.

Vertices A graph in the GIRG model consists of a set of n vertices $V = \{v_1, \dots, v_n\}$.

Degree Distribution β The weights follow a power-law distribution that has a power-law exponent $\beta > 2$. The property of a power-law distribution is that the fraction of weights with weight at least w is proportional to $w^{1-\beta}$.

Dimension d Each vertex has a position x_v in the ground space $\mathbb{X}^d$ with $\mathbb{X} \in \{\mathbb{T}, [0,1]\}$ and dimension $d \in \mathbb{N}$.

Average Degree c The parameter $c \in \mathbb{R}^+$ controls the expected average degree of the overall graph.

Temperature T The parameter $T \in [0,1]$ is the temperature which controls the binomial variant. As in the HRG model the model is called *binomial model* for $T > 0$ and *threshold model* for $T = 0$.
From these parameters we can sample a GIRG as follows. Each vertex has a position x_v on the ground space which is drawn uniformly and independently at random. Additionally each vertex has a weight w_v. The set of weights follows a power-law distribution where the density function of the probability distribution is given by

$$\rho(w) = (\beta - 1) \cdot w^{-\beta}. \tag{2.1}$$

The probability that two vertices $u, v \in V$ are adjacent is given for the binomial case as

$$p_{uv} = \min\left\{1, c\left(\frac{w_u w_v}{W \cdot \parallel x_u - x_v \parallel^d}\right)^{\frac{1}{T}}\right\} \tag{2.2}$$

where $W = \sum_{v \in V} w_v$ is the sum of all weights. As we can see, the weight controls the expected degree of the vertex because a higher weight yields higher connection probabilities and thus a higher vertex degree. In the threshold case with $T = 0$ the following equation describes the probability that two vertices are adjacent

$$p_{uv} = \begin{cases} 1 & \text{if } \parallel x_u - x_v \parallel \leq c\left(\frac{w_u w_v}{W}\right)^{\frac{1}{d}}, \\ 0 & \text{else.} \end{cases} \tag{2.3}$$

2.2.3 Relation to HRG

As mentioned previously the HRG model can be seen as a special case of the GIRG model. There exists a mapping from the HRG model to the GIRG model with dimension 1 where the weights and coordinates are given by

$$w_v := ne^{-r_u/2} \text{ and } x_v = \frac{\varphi_v}{2\pi}.$$

This mapping is a bijection and thus we can obtain a GIRG that is similar to the HRG and the other way round. The proof for this is shown in [Keu18]. We see that the HRG model and the 1-dimensional GIRG model are very similar. That is why our hope is that the approach of the HRG embedder also works well for the GIRG model.

2.3 Spring Embedder

An important embedding technique that is used in our algorithm is a *spring embedder* also called a *force-directed embedder* [BS03]. As the name suggests we apply forces to vertices to obtain an embedding with a low energy. At the beginning of the embedding process we start with a random position for each vertex. After that we repeat the following process until the configuration does not change any more. For every pair of vertices we calculate an attractive force that moves vertices that belong together closer to each other or a repulsive force that pushes vertices that are too close away from each other. After that these forces are summed up for each vertex and then applied by adjusting its position accordingly. When the configuration is stable or a certain number of iterations is reached, the process stops. This method is mostly used for the two-dimensional Euclidean space but can also be extended to higher-dimensional Euclidean spaces or non-Euclidean spaces like the hyperbolic space. But there are some complications in the hyperbolic space, which we discuss in Section 3.1.2.2.
There are several ways to define the forces between vertices. One example is to apply a repulsive force to each pair of vertices and an attractive force to each pair of vertices with an edge between them. This leads to good results because edges are typically short and non-edges are long and the vertices are uniformly distributed over the available space because of the repulsive forces. Another option is to calculate the forces such that the forces are in equilibrium when the distances in the embedding are proportional to the graph theoretical distances. This means that, e.g., the shortest path between each pair of vertices is calculated and a repulsive force is applied if their distance in the embedding is closer than their distance in the graph or an attractive force if the distance is greater than the distance in the graph. We use a similar approach where we estimate the distances between each pair of vertices and use these distances to calculate the forces. A disadvantage of spring embedders is that, without rather sophisticated enhancements, they only produce good results for very small graphs [Kob13] and so we cannot apply them to the graphs we deal with which are typically

very large. Another problem is that spring embedders struggle with heterogenous graphs because high degree vertices move loosely connected parts of the graph together [BFK21]. To avoid this problem we only use the spring embedder for high weight nodes and embed the remaining vertices based on that embedding.

2.4 Likelihood

We want the embedding to represent the graph structure, that is adjacent vertices are close and non-adjacent vertices are far apart. This is captured by the connection probability of two vertices in GIRGs, where the probability for an edge to exist depends on the distance between its endpoints. In particular, given an embedding in a weighted geometric space, we can look at each vertex pair, compute the probability for an edge to exist between them and compare that with the adjacency information in the graph. More precisely, the *likelihood* of an embedding measures the probability that an embedding matches the given graph. The goal of a maximum likelihood embedding is that all edges and non-edges that are induced by the embedding occur in the graph. The probability that this happens is given by the product of the probability of all edges and the complementary probability of all non-edges: $\Pi_{uv \in E} p_{uv} \cdot \Pi_{uv \notin E}(1 - p_{uv})$. When trying to maximize the likelihood, we can simplify this by taking the logarithm of that function which is valid since the logarithm is continuous and monotonous. The result is the *log-likelihood*: $\mathcal{L}\left(\{m\}_{i=1}^{n} | G\right) = \sum_{uv \in E} \log\left(p_{uv}\right) + \sum_{uv \notin E} \log\left(1 - p_{uv}\right)$ where m_i are the parameters of a vertex v_i in a model. A further simplification can be made by only considering one vertex v

$$\mathcal{L}(v) = \sum_{u \in \Gamma(v)} \log(p_{uv}) + \sum_{u \notin \Gamma(v)} \log(1 - p_{uv}). \tag{2.4}$$

This function can be used to measure the quality of the position of one vertex in the embedding. With this we can write the log-likelihood as $\mathcal{L}\left(\{m\}_{i=1}^{n} | G\right) = \frac{1}{2} \sum_{v \in V} \mathcal{L}\left(v\right)$ [BFKL16].

3 Embedding Algorithm

We have seen in Section 2.2.3 that HRGs and GIRGs are very similar. That is why we use an approach that already works well for the HRG model and apply it to the GIRG model. First we explain the basics of the HRG embedder, which we then transfer to the GIRG model.

3.1 Hyperbolic Embedding

The hyperbolic maximum likelihood embedder consists of three phases: parameter estimation, core embedding, and periphery embedding. We refer to the paper for a detailed description of the phases [BFKL16]. In the following we briefly summarize the aspects that are relevant for our approach.

3.1.1 Parameter Estimation

There are five parameters that can be estimated beforehand. These are the number of vertices n, the power-law exponent α, the temperature T and the radius of the disk R as well as the radial coordinates r_i for each vertex $v_i \in V$. The power-law exponent α is estimated with the algorithm [CSN07]. The temperature T is set to the fixed value of 0.1.

3.1.2 Core Embedding

The first step is to find a good embedding for the core. To be more specific, the vertices of the graph are partitioned into layers $L_i = \{v \mid 2^i \leq \deg(v) < 2^{i+1}\}$. The *core* of the graph is then defined as

$$C = \bigcup_{i \geq \frac{\log(n)}{2}} L_i.$$

It is very important that the embedding of the core is good otherwise we get a bad overall embedding because the second phase heavily depends on a good core embedding. A spring embedder as described in Subsection 2.3 is used to embed the core. The core contains all high degree vertices and forms a clique. That means the spring embedder can not use the edges and calculate repulsive forces for non-edges and attractive forces for edges because there are only edges and thus attractive forces. So another approach is needed. The forces for the spring embedder are calculated from the estimated distance between each pair of vertices based on their common neighborhood.

3.1.2.1 Estimating Angular Difference between Vertices

The angular difference between each pair of core vertices is approximated based on their common neighborhood. Vertices with small angular difference are more likely to be be adjacent and thus vertices with a large common neighborhood are more likely to have a small angular difference because they are adjacent to a lot of the same vertices. The resulting angular difference for two vertices u and v with common neighborhood of size c_{uv} is

$$\varphi\left(c_{uv}, r_u, r_v\right) = \Theta\left(1\right) \cdot c_{uv}^{1/(1-2\alpha)} \cdot \exp\left(-\frac{1}{2} r_u + \left(\frac{1}{2-4\alpha}\left(r_v - R\right)\right)\right).$$

To determine the constant factor the values for all pairwise distances are scaled such that their median is $\pi/2$. After the scaling all values greater than π are set to π.

3.1.2.2 Spring Embedder

As mentioned in Subsection 2.3 there are some restrictions with spring embedders in the hyperbolic space. Since the space in the hyperbolic space increases exponentially the only way for two vertices to get closer is to get closer to the origin. But a vertex that gets closer to the origin also gets closer to every other vertex in the graph. Because of that there are a lot of repulsive forces which lead to stable but bad embeddings. A solution for this is to use a disk in the hyperbolic space with only a small radius. Here the geometry is closer to the Euclidean geometry and the problems of the hyperbolic space do not influence the embedding process as much as with larger radii. This is exactly what is done when embedding the core because the vertices of the core are close to the origin.
To calculate the force between each pair of vertices u and v, the difference between the estimated angular difference $\varphi(c_{uv}, r_u, r_v)$ and the actual angular difference $\varphi_v - \varphi_u$ (where $0 \leq \varphi_u < \varphi_v \leq \pi$) is calculated. This is $\mathrm{err}(u, v) = (\varphi_v - \varphi_u) - \varphi(c_{u,v}, r_u, r_v)$. The force between two vertices is given by

$$F_u(v) = \begin{cases} -\mathrm{err}(u,v)^2 & \text{if } \mathrm{err}(u,v) \leq 0 \\ \mathrm{err}(u,v)^2 & \text{if } 0 < \mathrm{err}(u,v) \leq \frac{\pi}{2} \\ (\pi - \mathrm{err}(u,v))^2 & \text{if } \frac{\pi}{2} < \mathrm{err}(u,v) \leq \pi \end{cases}$$

To obtain the resulting force on a vertex u the forces $F_u(v)$ for all vertices $v \in C$ are summed up. The spring embedder is run five times with different initial coordinates and then the best embedding is chosen.

3.1.3 Periphery Embedding

To embed the remaining vertices the layers are used. For each layer L_i starting with the highest layer the position of each vertex in layers L_j $(j \geq i)$ is optimized by its log-likelihood. This process is done $\log n$ times per layer. If a vertex is embedded for the first time its initial position is estimated by a weighted average of all neighbors. This ensures that neighbors with a large radius have more influence on the position and hence are nearer to the new vertex. For a vertex v the initial position is determined by

$$\varphi_v = \arctan\left(\frac{\sum_{i=1}^{k} \exp\left(r_{u_i}\right) \cdot \sin\left(\varphi_{u_i}\right)}{\sum_{i=1}^{k} \exp\left(r_{u_i}\right) \cdot \cos\left(\varphi_{u_i}\right)}\right)$$

If a vertex is already embedded its position used. The position of a vertex is updated by sampling $\mathcal{O}(\log n)$ positions around the initial position and the position with the smallest log-likelihood is chosen. This is efficient because the log-likelihood can be computed efficiently with a geometric data structure by only taking the neighbors and non-neighbors with a low distance into account and roughly approximating the non-neighbors with a large distance.

3.2 Weighted Geometric Embedding

Our algorithm consists of three steps. In the first step we estimate the model parameters of the input graph. In the second step we embed the core of the graph. In the third step we embed the remaining vertices.

3.2.1 Parameter Estimation

We start by estimating the model parameters n, β, c, T and d (see Section 2.2.2). The power-law exponent and the temperature are estimated as in the HRG embedder.

Dimension d The estimation of the dimension is part of another paper and can be computed algorithmically [FGKS23a].

Weights w_v We assume that the degree of the vertices in the given graph follows a power-law distribution. The expected degree of a vertex in the GIRG model matches the weight up to constants. For the sake of simplicity, we set the weight of the vertex to its degree, i.e., $w_v = \deg(v)$.

Expected Average Degree Parameter c The parameter c controls the expected average degree of the graph. The estimation of c from the average degree of the graph is explained in [Blä+22].

3.2.2 Core Embedding

In this subsection we only look at high-degree vertices in layers L_i with $i \geq \frac{\log n}{2}$, which form the core. We want to find a good embedding for them before we embed the other vertices. To achieve this, we estimate the distance for each pair of core vertices based on the size of their common neighborhood. Then we use a spring embedder to embed the vertices accordingly. We refer to this part of the algorithm as the *core phase* in the following.

3.2.2.1 Expected Common Neighborhood

We only perform this calculation assuming the threshold model and a torus as ground space. We have found out experimentally that this also works well for the cube and the binomial model (see Figure 3.2c). We want to calculate the expected distance between two vertices u and v based on the size of their common neighborhood. To do this we use the model and calculate the expected size of the common neighborhood of u and v assuming we know their distance. Then we can solve the resulting equation for the distance and calculate the distance from the common neighborhood which we already know. To this end we need the probability $p_g(w_a)$ that a third vertex a is connected to u and v when u and v have a fixed distance $\|x_u - x_v\|$ between them and a has a weight w_a. The integral of the probability $p_g(w)$

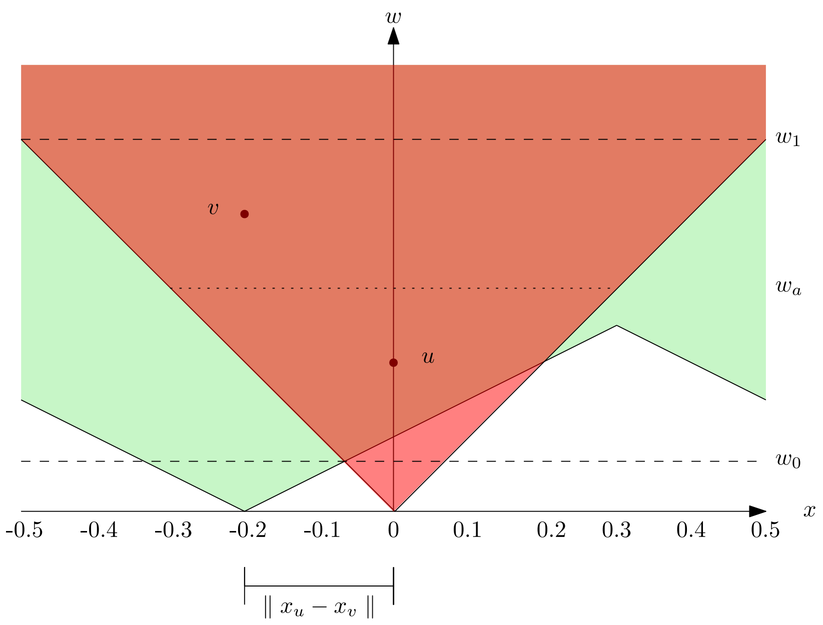


Figure 3.1: Visualization of a 1-dimensional torus in the threshold model. The coordinate is on the x-axis and the weight on the y-axis. The green colored area covers all vertices that are connected to the vertex v and all vertices in the red colored area are connected to u. Vertices that have a weight below w_0 cannot be connected to both vertices at the same time. Vertices that have a weight higher than w_1 are in any case connected to both, u and v. For all vertices with a weight w_a between w_0 and w_1 the probability to be connected to both, u and v, is the length of the overlapping area (here the dotted line). We can see that the linear approximation fits well because we approximate the width of the triangle which increases linearly but changes the slope at one point.

multiplied by the density function of the power-law distribution is then the expected size of the common neighborhood. We approximate p_g with a linear function. For this, we compare two weights w_0 and w_1, such that $p_g(w_0) = 0$ and $p_g(w_1) = 1$. This can be seen in Figure 3.1. Then we can interpolate the function p_g between these two points linearly. Note that by the definition of the model (see Equation 2.3) a is in the neighborhood of u and v if both inequalities hold

$$\|x_v - x_a\| \leq \frac{c}{W^{\frac{1}{d}}} \cdot (w_v \cdot w_a)^{\frac{1}{d}} = \Delta(v, a), \tag{3.1}$$

$$\|x_u - x_a\| \leq \frac{c}{W^{\frac{1}{d}}} \cdot (w_u \cdot w_a)^{\frac{1}{d}} = \Delta(u, a) \tag{3.2}$$

where $\Delta(u, v)$ is the maximum distance between u and v such that u and v are connected. First, we calculate the weight w_1 such that $p_g(w_a) = 1$ for all $w_a \geq w_1$. In this case both equations need to hold. Without loss of generality let $w_u \leq w_v$. If we have a weight w_a such that Equation 3.2 is true for all possible distances $\|x_u - x_a\|$, then Equation 3.1 is also true because v has a higher weight than u. So we only need to solve Equation 3.2 for this.

Let d_{max} be the maximum distance on the torus, which is 0.5 when using the L_∞-norm and $0.5\sqrt{d}$ when using the L_2-norm. Thus, Equation 3.2 holds when w_a is sufficiently large such that $d_{max} \leq \Delta(u, a)$, which is equivalent to

$$w_a \geq \frac{d_{max}^d W}{c^d w_u} =: w_1.$$

Now we calculate the weight w_0. Note that if $\Delta(v, a) + \Delta(u, a) < \|x_v - x_u\|$ then u and v are so far apart that they cannot connect to both of them simultaneously. From this follows

$$\begin{aligned} \Delta(u, a) + \Delta(v, a) &\geq \|x_v - x_u\| \\ \Leftrightarrow \frac{c}{W^{\frac{1}{d}}} w_a^{\frac{1}{d}} \left(w_u^{\frac{1}{d}} + w_v^{\frac{1}{d}}\right) &\geq \|x_v - x_u\| \\ \Leftrightarrow w_a &\geq \frac{W \|x_v - x_u\|^d}{c^d \left(w_u^{\frac{1}{d}} + w_v^{\frac{1}{d}}\right)^d} =: w_0. \end{aligned}$$

Now we approximate the probability of a being connected to u and v with a linear function. That means we can write $p_g(w) \approx \tilde{p_g}(w) = Aw + B$. To solve this we have the following equations

$$\begin{aligned} p_g(w_1) &= 1 = Aw_1 + B \\ p_g(w_0) &= 0 = Aw_0 + B \end{aligned}$$

The solution is

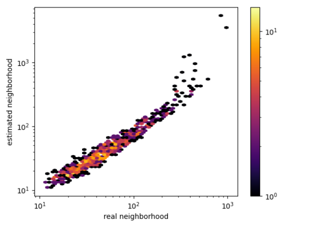
$$\begin{aligned} A &= \frac{1}{w_1 - w_0} = \frac{c^d \tilde{w}}{W \cdot \left(\frac{d_{max}^d \tilde{w}}{w_u} - \|x_u - x_v\|^d\right)} \\ B &= -\frac{w_0}{w_1 - w_0} = -\frac{\|x_u - x_v\|^d}{\frac{d_{max}^d \tilde{w}}{w_u} - \|x_u - x_v\|^d} \end{aligned}$$

with $\tilde{w} = \left(w_u^{\frac{1}{d}} + w_v^{\frac{1}{d}}\right)^d$. To get the expected size of the common neighborhood we have to integrate p_g multiplied with the density function of the probability distribution $\rho(w)$ of the weights (given in Equation 2.1). The expected size c_{uv} of the common neighborhood is

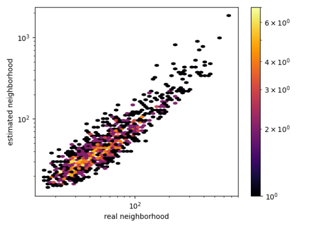
$$\begin{aligned} c_{uv} &= \int_0^\infty \rho(w) p_g(w) dw \\ &= \int_{w_0}^{w_1} \rho(w) p_g(w) dw + \int_{w_1}^\infty \rho(w) dw \\ &\approx \int_{w_0}^{w_1} (\beta - 1) w^{-\beta} (Aw + B) dw + \int_{w_1}^\infty (\beta - 1) w^{-\beta} dw \\ &= -A\frac{\beta - 1}{\beta - 2} w_1^{2-\beta} + Bw_1^{1-\beta} + A\frac{\beta - 1}{\beta - 2} w_0^{2-\beta} - Bw_0^{1-\beta} + w_1^{1-\beta} \\ &= A\frac{\beta - 1}{\beta - 2}\left(w_0^{2-\beta} - w_1^{2-\beta}\right) + B\left(w_0^{1-\beta} - w_1^{1-\beta}\right) + w_1^{1-\beta}. \end{aligned}$$

We can see that this is a good estimation of the expected neighborhood in Figure 3.2. Now we have to solve this equation for the distance $\|x_u - x_v\|$ between u and v which is given by

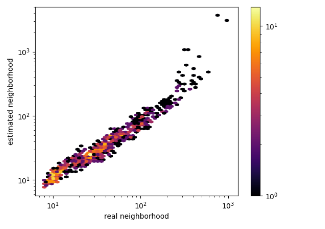
$$k_1(c_{uv}) \approx k_2(c_{uv}) \|x_u - x_v\|^d + k_3(c_{uv}) \|x_u - x_v\|^{d(2-\beta)} \tag{3.3}$$



(a) Graph with dimension 1 and torus as ground space. The linear approximation works quite well.



(b) Graph with dimension 3 and torus as ground space. The linear approximation works also well for higher dimensions.



(c) Graph with dimension 1 and cube as ground space. The linear approximation works also well for the cube.

Figure 3.2: This plot shows the expected neighborhood on the y-axis and the real neighborhood on the x-axis. We sampled a GIRG with 20000 vertices, average degree of 10, temperature 0.1 and power-law exponent 2.5 which resulted in 43-44 core vertices. Then we calculated the expected size of the common neighborhood for all core vertex pairs as described and compared them to their real common neighborhood. The distances are calculated with the L_2-norm. The color indicates the number of vertices. The values for the neighborhood are scaled such that the squared deviation is minimized.

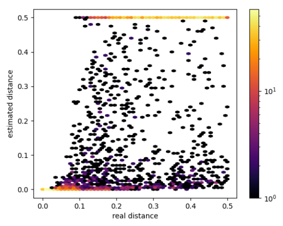
with parameters

$$
\begin{aligned}
k_1(c_{uv}) &= \frac{d_{max}^d \tilde{w}}{w_u}\left(c_{uv} + \left(\frac{d_{max}^d W}{c^d w_u}\right)^{1-\beta}\left(\frac{\beta-1}{\beta-2}-1\right)\right)\\
k_2(c_{uv}) &= c_{uv}\\
k_3(c_{uv}) &= \left(\frac{\beta-1}{\beta-2}-1\right)\left(\frac{W}{c^d \tilde{w}}\right)^{1-\beta}.
\end{aligned}
$$

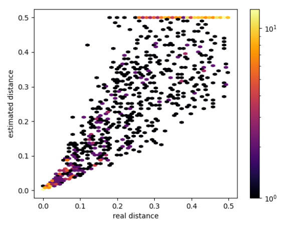
Note that this expression is tedious to solve exactly. Therefore, we approximate a solution instead. Recall that $\|x_u - x_v\| \leq \frac{1}{2}$. Thus raising the distance to the $d(2-\beta)$, which is negative as we assume $\beta > 2$ yields larger values. On the other hand, raising small distances to the power of $d > 0$ yields smaller values, which is why we neglect the term $k_2(c_{uv}) \|x_u - x_v\|^d$. Thus, we estimate the distance between two points as

$$
\text{dist}_{opt}(u,v) = \left(\frac{k_1(c_{uv})}{k_3(c_{uv})}\right)^{\frac{1}{d(2-\beta)}}.
$$

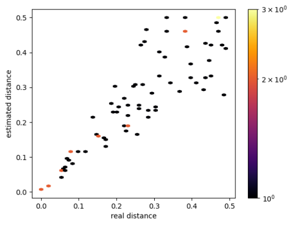
The expected distances are than scaled such that their median is $d_{max}/2$ and all distances greater than d_{max} are set to d_{max}. An example plot of this estimation with different power-law exponents can be seen in Figure 3.3. We note that for small power-law exponents ($\beta \leq 2.3$) the accuracy of the estimation declines as shown in Figure 3.3a. A reason for this is that with smaller power-law exponent there is not as much information about the neighborhood as with higher power-law exponents. A comparison of the L_∞ to the L_2-norm can be seen in Figure 3.4. We can see that the result of the L_2-norm is very similar to the L_∞-norm.
Because the variance of the distance is quite large we can use an approach other than computing the distance directly. Assuming that the common neighborhood is monotonous in the distance, we can use binary search on the interval $[0, 0.5]$ to estimate the distance. We



(a) Power-law exponent 2.1 and 74 core vertices.



(b) Power-law exponent 2.5 and 44 core vertices.



(c) Power-law exponent 2.9 and 13 core vertices.

Figure 3.3: This plot shows the expected distance on the y-axis and the real distance on the x-axis. We sampled a GIRG with 20000 vertices, average degree of 10, temperature 0.1 and dimension 1. We calculated the expected distance for all core vertex pairs as described and compared them to their real distance. The distances are calculated with the L_2-norm.

start with an initial value of 0.25 and add or subtract in iteration i the value $1/2^{i+2}$ until the expected common neighborhood is close to the real neighborhood in the graph. Surprisingly, approximation and binary search yield nearly identical results as can be seen in Figure 3.4. This means that the estimation for the expected common neighborhood works well and we can use it in the algorithm.

3.2.2.2 Spring Embedder

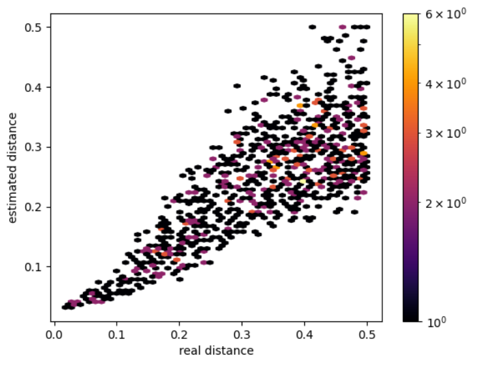
Now we can use the estimated distances from the previous subsection to embed the vertices with a spring embedder as described in Section 2.3. In the first step we assign every vertex a random initial position uniformly in the ground space. After that we calculate in every step for each pair of vertices u and v the difference between the current distance $\|x_u - x_v\|$ and the desired distance $\text{dist}_{opt}(u, v)$ denoted by

$$\text{err}(u, v) = \|x_u - x_v\| - \text{dist}_{opt}(u, v).$$

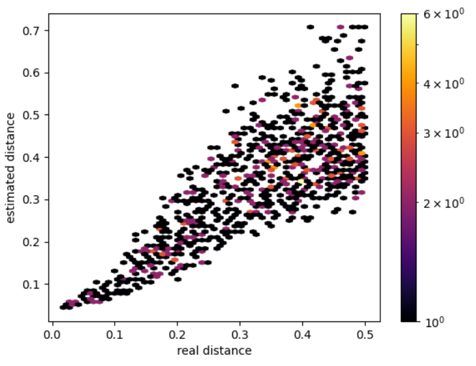
If the vertices are further apart than they should be ($\text{err}(u, v) \geq 0$) we apply attractive forces to pull the vertices towards each other. If the vertices are closer than they should be ($\text{err}(u, v) < 0$) we apply repulsive forces to push them further apart. If two vertices on the torus are very far apart but should be very near the error function is close to d_{max}. That means that the force for this vertex is very strong for one direction. However, because of the geometry it does not matter in which direction a vertex moves because the distance to the other vertex is decreased in both directions. So we do not want a strong force for too large distances. To achieve this we decrease the force again when a distance of $d_{max}/2$ is reached. For a given vertex v the force to vertex u is calculated as

$$F_v(u) = \begin{cases} -\text{err}(u, v)^2 \cdot (x_u - x_v) & \text{if } -d_{max} < \text{err}(u, v) \leq 0 \\ \text{err}(u, v)^2 \cdot (x_u - x_v) & \text{if } 0 < \text{err}(u, v) \leq d_{max}/2 \\ \left(\frac{1}{2} - \text{err}(u, v)\right)^2 \cdot (x_u - x_v) & \text{if } d_{max}/2 < \text{err}(u, v) \leq d_{max}. \end{cases}$$

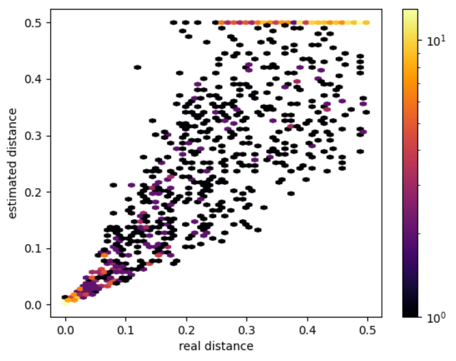
However, for the cube this is different. We do not decrease the force there because we do not have the wrap-around of the torus.
We note that, in the original paper, the spring embedder moved vertices on a circle, which is a 1-dimensional movement. An attractive force moves v in the direction where the distance to



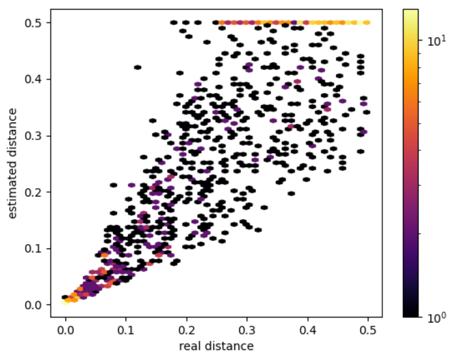
(a) L_∞-norm, dimension 2 and estimation.



(b) L_2-norm, dimension 2 and estimation.



(c) L_2-norm, dimension 1 and estimation.



(d) L_2-norm, dimension 1 and binary-search.

Figure 3.4: This plot shows the expected distance on the y-axis and the real distance on the x-axis. We sampled a GIRG with parameters 20000 vertices, average degree of 10, temperature 0.1 and power-law-exponent 2.5 which resulted in 44 core-vertices. We calculated the expected distance for all core vertex pairs as described and compared them to their real distance.

the target is smaller. A repulsive force moves a vertex in the opposite direction. For higher-dimensional spaces this is more involved. If we use the gradient of the L_∞-norm to calculate the direction we would only move in one dimension. This is why we use the gradient of the L_2-norm to determine the direction of the force. On the torus we compute the direction such that $|x_{ui} - x_{vi}| < 0.5$ $(1 \le i \le d)$. That means we always take the shortest way on the torus to the other vertex. The resulting overall force applied to a vertex v is $F_v = \sum_{u \in C \setminus \{v\}} F_v(u)$. The quality of the embedding is measured with a score

$$S = \sum_{u \in C} \sum_{v \in C \setminus \{u\}} |F_u(v)|.$$

We run the spring embedder five times with different initial coordinates and choose the best embedding from them. Regarding the running time, note that the core consists of $\Theta(n^{(3-\beta)/2})$ vertices [FGKS23b]. Since a single iteration of the spring embedder considers forces between all vertex pairs, this yields a running time of $\Theta(n^{3-\beta})$ per iteration. Allowing $n^{\beta-2}$ iterations yields a linear running time in total.

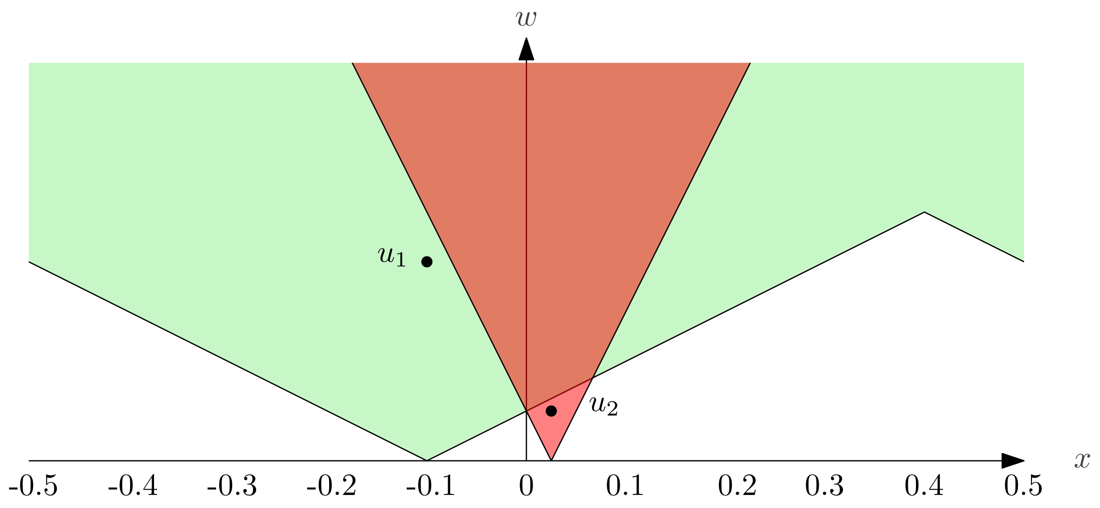


Figure 3.5: Visualization of a 1-dimensional torus in the threshold model. The coordinate is on the x-axis and the weight on the y-axis. All vertices that are in the green colored area are connected to the vertex u_1 and all vertices in the red colored area are connected to u_2. We want to find a good position for a vertex v which has u_1 and u_2 as neighbors. It is only possible for v to be connected to both if v is placed in the area where the green and red area overlap. The low weight vertex u_2 has much more influence for this position than the high weight vertex u_1. The vertex v can have a nearly arbitrary position on the torus to be connected to the high weight vertex. Thus high weight neighbors have a smaller influence on the position of a newly embedded vertex than low weight neighbors.

3.2.3 Periphery Embedding

After we have embedded the core vertices we have to embed the remaining vertices. These are the vertices in layers $L_i = \{v \mid 2^i \leq \deg(v) < 2^{i+1}\}$ for $i < \log n/2$. We start this process from the highest to the lowest layer. For each vertex we first calculate an initial position. To measure the quality of a position we use the log-likelihood for one vertex as given in Equation 2.4. We refer to this part of the algorithm as *periphery phase* in the following. Since we are using the logarithm of the probability for the log-likelihood we have to deal with the probability 1. We sum the complementary probabilities for the non-edges. If a non-edge has a probability 1 we have a log-likelihood of log 0 which is undefined. That is why we use the use the Fermi-Dirac-Equation to approximate the probability in Equation 2.2. This equation eliminates the minimum and produces results in $(0, 1)$. The Fermi-Dirac-Equation is given by

$$p_{uv} \approx \frac{1}{1 + \frac{1}{c}\left(\frac{w_u w_v}{W\|x_u - x_v\|^d}\right)^{-\frac{1}{T}}} =: \tilde{p_{uv}}. \tag{3.4}$$

Initial Position

We calculate the initial position for a vertex v as the weighted average of the already embedded neighbors $\Gamma'(v)$. We want the vertex v to be connected to all other already embedded neighbors. The connection probability depends on the the weight of the vertices and the distance between them. This means we can have a larger distance to vertices with a high weight but we need a shorter distance to vertices with a low weight. Since we want to be closer to vertices with a small weight, we use the inverted weight for the weighted average. This is shown in Figure 3.5. We define the initial position of v to be

$$x_v = \frac{\sum_{u \in \Gamma'(v)} \frac{1}{w_u} x_u}{\sum_{u \in \Gamma'(v)} \frac{1}{w_u}}. \tag{3.5}$$

This formula works well on a cube to get the weighted average position but on a torus it is not clear how we can get a weighted average position. The wrapping makes it hard to calculate a weighted average position. Now we want to move the vertices from their initial position to a better position that is a position with a better log-likelihood. There we have two different options. For the first option we use a heuristic approach and sample different positions around the initial position. For the second option we use the gradient of the log-likelihood to optimize the position.

Option 1: Sampling After we have computed a good initial position for a given vertex, we can sample $\log(n)^d$ different random positions distributed uniformly in a d-dimensional cube around the initial position with length λ for the vertex v. After that we compute the log-likelihood for each position and we take the coordinates of the position with the highest log-likelihood. To this point our approach is similar to the HRG embedder. However, we do this process 3 times while decreasing the sample radius λ. We halve the sample radius in every iteration. This ensures that we are close to a local maximum for the log-likelihood of the vertex v.

Option 2: Gradient Ascent We can also use gradient ascent to find a local maximum of the function $\mathcal{L}(v)$. To this end we use the derivative of the function $\mathcal{L}(v)$ that is $\frac{\partial \mathcal{L}(v)}{\partial x_v}$. We start with the initial position $x_{v,0}$ and take the derivative to compute a better position. This position is given by $x_{v,i+1} = x_{v,i} + \gamma \frac{\partial \mathcal{L}(v)}{\partial x_v}$ where γ is the *learning rate*. The derivative is given by

$$\begin{aligned}
\frac{\partial \mathcal{L}(v)}{\partial x_v} &= \frac{\partial \left(\sum_{u \in \Gamma(v)} \log(p_{uv}) + \sum_{u \notin \Gamma(v)} \log(1 - p_{uv}) \right)}{\partial x_v} \\
&= \sum_{u \in \Gamma(v)} \frac{\partial \log(p_{uv})}{\partial x_v} + \sum_{u \notin \Gamma(v)} \frac{\partial \log(1 - p_{uv})}{\partial x_v} \\
&= \sum_{u \in \Gamma(v)} \frac{\partial \log(p_{uv})}{\partial p_{uv}} \frac{\partial p_{uv}}{\partial x_v} + \sum_{u \notin \Gamma(v)} \frac{\partial \log(1 - p_{uv})}{\partial p_{uv}} \frac{\partial p_{uv}}{\partial x_v} \\
&= \sum_{u \in \Gamma(v)} \frac{1}{p_{uv}} \frac{\partial p_{uv}}{\partial \|x_u - x_v\|} \frac{\partial \|x_u - x_v\|}{\partial x_v} \\
&\quad - \sum_{u \notin \Gamma(v)} \frac{1}{1 - p_{uv}} \frac{\partial p_{uv}}{\partial \|x_u - x_v\|} \frac{\partial \|x_u - x_v\|}{\partial x_v}.
\end{aligned}$$

Here we have to calculate two derivatives, the probability derived with respect to the distance and the distance derived with respect to the position. The probability given in Equation 2.2 has a minimum. That is why we have to make a case differentiation. The minimum evaluates to 1 if $(w_u w_v/(W\,\|x_u - x_v\|^d))^{1/T} \geq 1$. The derivate of this is 0. That means we only look at vertex pairs with $p_{uv} < 1$ when calculating the gradient. The derivate of the probability is given by

$$\begin{aligned}
\frac{\partial p_{uv}}{\partial \|x_u - x_v\|} &= \frac{\partial c\left(\frac{w_u w_v}{W\|x_u - x_v\|^d}\right)^{\frac{1}{T}}}{\partial \|x_u - x_v\|} \\
&= c\left(\frac{w_u w_v}{W}\right)^{\frac{1}{T}} \frac{\partial \|x_u - x_v\|^{-\frac{d}{T}}}{\partial \|x_v - x_u\|} \\
&= -\frac{d}{T} c\left(\frac{w_u w_v}{W}\right)^{\frac{1}{T}} \|x_u - x_v\|^{-\frac{d}{T}-1} \\
&= -\frac{d}{T} p_{uv} \|x_v - x_u\|^{-1}.
\end{aligned}$$

If we plug this in the first equation we get

$$\begin{aligned}
\frac{\partial \mathcal{L}(v)}{\partial x_v} &= -\frac{d}{T} \sum_{u \in \Gamma(v), p_{uv} < 1} \frac{1}{p_{uv}} p_{uv} \|x_u - x_v\|^{-1} \frac{\partial \|x_u - x_v\|}{\partial x_v} \\
&\quad + \frac{d}{T} \sum_{u \notin \Gamma(v), p_{uv} < 1} \frac{1}{1 - p_{uv}} p_{uv} \|x_u - x_v\|^{-1} \frac{\partial \|x_u - x_v\|}{\partial x_v} \\
&= -\frac{d}{T} \sum_{u \in \Gamma(v), p_{uv} < 1} \|x_u - x_v\|^{-1} \frac{\partial \|x_u - x_v\|}{\partial x_v} \\
&\quad + \frac{d \cdot c}{T} \sum_{u \notin \Gamma(v), p_{uv} < 1} \frac{(w_u w_v)^{\frac{1}{T}} \|x_u - x_v\|^{-1}}{\left(W\|x_u - x_v\|^d\right)^{\frac{1}{T}} - c(w_u w_v)^{\frac{1}{T}}} \frac{\partial \|x_u - x_v\|}{\partial x_v}.
\end{aligned}$$

In the last step we used the definition of the probability p_{uv} to simplify the equation. If we use the L_∞-norm the derivate $\frac{\partial\|x_u - x_v\|}{\partial x_v}$ is given by

$$\frac{\partial \|x_u - x_v\|_\infty}{\partial x_v} = \begin{pmatrix} 0 \\ \vdots \\ \pm 1 \\ \vdots \\ 0 \end{pmatrix}.$$

The 1 is on the i-th position where $\min\{|x_{uj} - x_{vj}|, 1 - |x_{uj} - x_{vj}|\} \leq \min\{|x_{ui} - x_{vi}|, 1 - |x_{ui} - x_{vi}|\}$ for all $1 \leq j \leq d$. The 1 is negative if $|x_{ui} - x_{vi}| \leq 1 - |x_{ui} - x_{vi}|$ and $x_{ui} \geq x_{vi}$ or if $|x_{ui} - x_{vi}| > 1 - |x_{ui} - x_{vi}|$ and $x_{ui} < x_{vi}$. Otherwise the 1 is positive.
If we use the L_2-norm instead this term becomes

$$\frac{\partial \|x_u - x_v\|_2}{\partial x_v} = \frac{1}{\|x_u - x_v\|_2} \begin{pmatrix} \pm \|x_{u1} - x_{v1}\|_2 \\ \vdots \\ \pm \|x_{ud} - x_{vd}\|_2 \end{pmatrix}.$$

The i-th entry is negative if $|x_{ui} - x_{vi}| < 1 - |x_{ui} - x_{vi}|$ and $x_{ui} > x_{vi}$ or $|x_{ui} - x_{vi}| \geq 1 - |x_{ui} - x_{vi}|$ and $x_{ui} \geq x_{vi}$. On the cube these derivatives are easier, for the L_∞-norm this is

$$\frac{\partial \|x_u - x_v\|_\infty}{\partial x_v} = \begin{pmatrix} 0 \\ \vdots \\ \pm 1 \\ \vdots \\ 0 \end{pmatrix}.$$

where the 1 is on the i-th position where $|x_{ui} - x_{vi}| = \max_{1 \leq j \leq d} |x_{uj} - x_{vj}|$. The 1 is positive if $x_{ui} < x_{vi}$ and otherwise negative.
The derivate of the L_2-norm for the cube looks like the following

$$\frac{\partial \|x_u - x_v\|_2}{\partial x_v} = \frac{1}{\|x_u - x_v\|_2} \begin{pmatrix} x_{v1} - x_{u1} \\ \vdots \\ x_{vd} - x_{ud} \end{pmatrix}.$$

We stop the gradient ascent if the gradient is smaller than a threshold epsilon or if we exceed $\mathcal{O}(\log(n))$ iterations.
If we calculate the log-likelihood as given in this equation we have to consider all other vertices in the graph. The running time for this is then in $\mathcal{O}\left(n^2\right)$. To obtain a linear running time, we have to calculate the log-likelihood more efficiently.

Computing the Log-Likelihood Efficiently

To compute the log-likelihood function, which is defined in Equation 2.4, we need $\Omega(n)$ time, which is too much. So we need to speed up the computation. The first term of the log-likelihood $\sum_{u \in \Gamma(v)} \log(p_{uv})$ can computed in $\mathcal{O}(|E|)$ for all vertices $v \in V$. With high probability $|E| = \Theta(n)$ [Keu18]. The previous term can be computed in linear time for all vertices and thus in amortized constant time for each vertex. This leaves the second Term $\sum_{u \notin \Gamma(v)} \log(1 - p_{uv})$. Our goal is to speed up the computation by ignoring negligible summands, i.e., the ones that are close to zero. Note that these are the ones where p_{uv} is close to zero. This happens when there is a large distance between two vertices and the product of their weights is small. We use the Fermi-Dirac-Equation for the probability as given in Equation 3.4

$$\tilde{p_{uv}} = \frac{1}{1 + \frac{1}{c}\left(\frac{w_u w_v}{W \cdot \|x_u - x_v\|^d}\right)^{-\frac{1}{T}}}.$$

For small weights and a large distance the denominator is very large and thus the probability is small. This means the term $\log(1 - p_{uv})$ is close to 0 and does not contribute to the log-likelihood. This means we can either ignore or coarsely approximate vertices with large distances and small weights. An efficient implementation can be realized through a spatial data structure (as in the original paper [BFKL16]). For this, it is necessary to define more precisely when a node is far enough away and a weight is small enough.

3.2.4 Algorithm

We are now ready to combine the previously described steps into one algorithm. First we estimate the necessary parameters and embed the core of the graph as explained in Section 3.2.1 and Section 3.2.2. Then we sort the remaining vertices in layers and iterate over the layers L_i

with $i < \log n/2$ in decreasing order. The following process is repeated $\log n$ times per layer. We look at every vertex $v \in \bigcup_{j \geq i} L_j$. If v already has a position from previous iterations, we keep it. Otherwise, we compute an initial position as described above. Then we adjust v's positions using gradient ascent or sampling, this optimizes the log-likelihood of every vertex. As a result of optimizing vertices from all layers not only the vertices with larger weight have an influence on the position of a vertex but also the vertices with smaller weight can affect their position.

4 Experimental Evaluation

In this section we want to see how well the embedding algorithm performs on different graphs and how their properties, such as the dimension or the power-law exponent, influence the quality of the embedding. We also want to know what impact on the algorithms performance each part of the algorithm has and what parts need to be improved in the future. Since the algorithm consists of two parts, the core phase and the periphery phase, we want to evaluate the performance of each part individually. Subsequently we analyze the algorithm as a whole. In particular, we want to address the following questions:

1. What is the overall quality of the produced embedding?
2. How do different graph properties influence the overall embedding?
3. Which of the two parts of the algorithm is the bottleneck when the quality deteriorates?
4. How do different graph properties influence the core phase?
5. How do different graph properties influence the periphery phase?

4.1 Evaluation Process

We evaluate the algorithm by first generating a GIRG, that means we already have an embedding for this graph, the ground truth embedding. Then we run the algorithm with the vertices and edges of the generated graph to get the embedding from our algorithm. We do not perform the estimation of the graph properties here but take the original values instead except for the weights, which are estimated. The reason for this is that we do not want to accumulate the errors from the parameter estimation. Then we have the positions of the vertices from the original generated GIRG and the positions from the algorithm. Now we want to know how close the quality of the embedding of our algorithm is to the ground truth. We use the log-likelihood which is defined in Equation 2.4 to measure the quality of the embedding. We have two graphs, the input graph G and the output graph $G' = \mathcal{A}(G)$ and we compare the log-likelihood by taking their quotient which we call *log-likelihood quotient* which is given by

$$\mathcal{L}_{compare} = \frac{\mathcal{L}(G')}{\mathcal{L}(G)}.$$

If we have a value of 1, the quality of the embedding of our algorithm is as good as the ground truth. Our goal is to get as close to the value 1 as possible. If the value is below 1, the generated embedding is more likely than the ground truth and if the log-likelihood quotient is higher than 1, the ground truth embedding is more likely. We sampled five graphs for each parameter combination in order to have a statistical relevant result. We use box plots to present the results of our evaluation. The aggregated results are represented by a box with a black line, whiskers and outliers. The black line is the median of the samples, the box contains all values between the 25th percentile and the 75th percentile. The whiskers cover all values between

the minimum value and the maximum value. The length of the whiskers is 1.5 IQR. All points outside of them are considered as outliers.
Another comparison to the ground truth embedding can be done with *corresponding-coordinates-plots*. A corresponding-coordinates-plot can be seen in Figure 4.2. There we plot for every vertex the coordinate of the ground truth embedding against the coordinate of the algorithms embedding. In the following the coordinates of the algorithms embedding are on the y-axis and the coordinates of the ground truth embedding are on the x-axis for the corresponding-coordinates-plots. If the plot is a line (with an optional cyclic shift for the torus) the embedding of the algorithm is the same as the ground truth. For higher dimensions it is possible that the algorithm swaps axes, e.g., the x_1-axis of the ground truth embedding and the x_2-axis of the algorithms embedding correlate and the x_2-axis of the ground truth embedding and the x_1-axis of the algorithms embedding correlate. That is why we draw a corresponding-coordinates-plot for each combinations of dimensions. If there is one corresponding-coordinates-plot in each row and column where the coordinates correlate we have a good embedding.

4.2 Degrees of Freedom

There are several options and graph properties which can influence the algorithms performance. We grouped them in four different categories in order to evaluate their influence on the embedding.

Graph	Ground Space	Core Embedding	Periphery Embedding
number of vertices	torus or cube	core distances: estimated or ground truth	core embedding: phase 1 embedding or ground truth
average degree	norm: L_2 or L_∞	initial positions: ground truth or random	initialization: estimates or ground truth
power-law-exponent	dimension		optimization method: sampling or gradient ascent
temperature			

4.3 Overall Quality

In this section we want to answer the question how good the overall quality of the algorithm is and which influence the graph properties have. We tested property combinations with different dimensions, power-law exponents, temperatures and ground spaces. The results are shown in Figure 4.1. In Figure 4.2 are corresponding-coordinates-plots for two examples of embeddings from the algorithm. We can clearly see that our embedding is close to the ground truth that means that the algorithms performance is pretty good which answers question 1 about the overall quality. The embedding is near the desired value of 1 when using the cube as ground space for high power-law exponents and temperature 0.1. With the results of this experiment we can answer question 2, which graph properties influence the algorithms performance. If we compare the ground spaces we notice that the algorithm performs better if the ground space is a cube with temperature 0.1. For a higher temperature the difference is more balanced. There are also some other things we can notice. We notice that for temperature $T = 0.1$ the quality decreases with increasing dimension. We also investigate

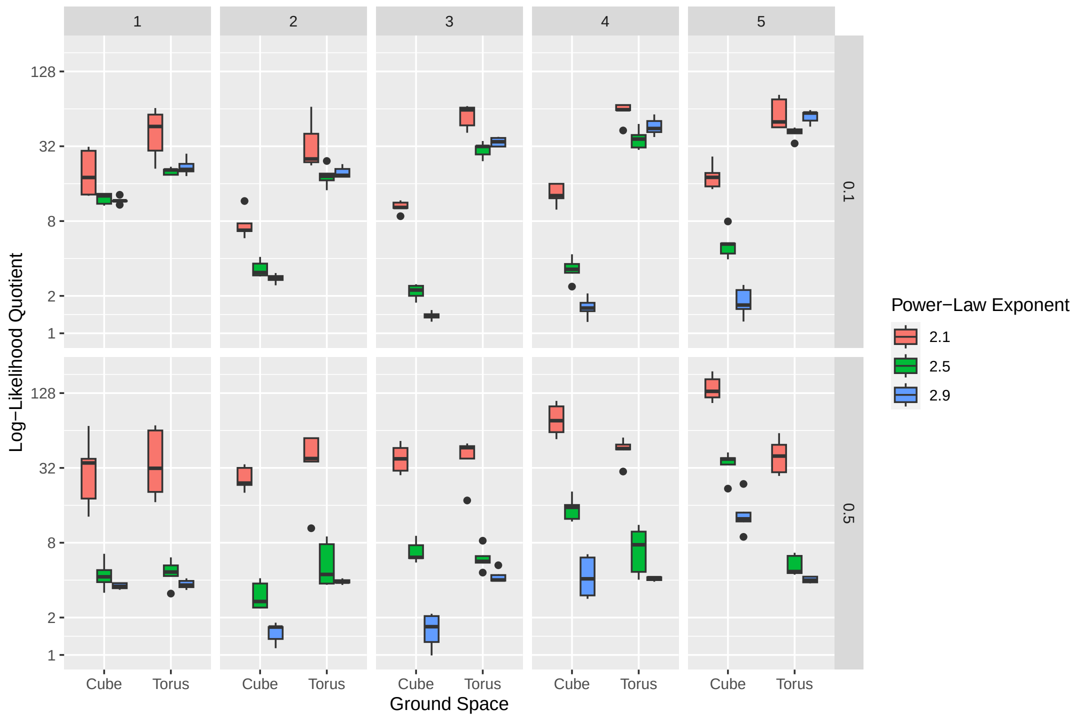
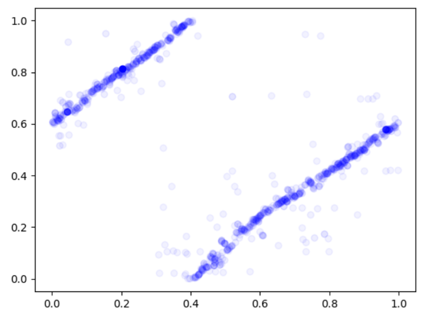


Figure 4.1: Experiments for the overall quality. We sampled graphs with 1000 vertices and an average degree of 10. We run the whole algorithm (including the core phase and the periphery phase) with different power-law exponents, dimensions, temperatures and ground spaces. The columns represent the dimension starting with dimension 1 to 5. The rows represent different temperatures.


Embedding of a graph with dimension 1.

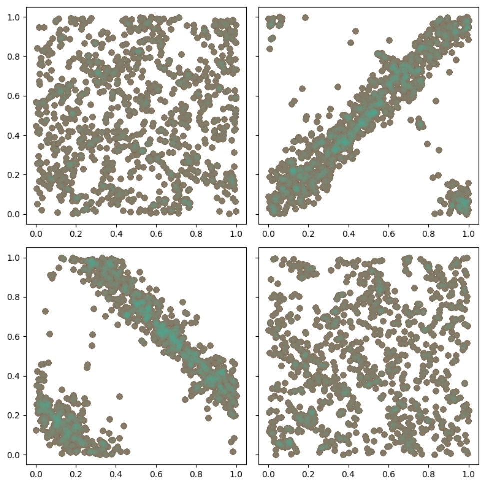

Embedding of a graph with dimension 2.

Figure 4.2: Corresponding-coordinates-plots for embeddings produced from the whole algorithm. We sampled a GIRG with 1000 vertices, average degree of 10, power-law exponent 2.9 and a temperature of 0.1 and a torus as ground space.

which part of the algorithm has difficulties with higher dimensions. The quality is bad for power-law exponent $\beta = 2.1$ but gets better for higher power-law exponents. As we saw in Section 3.1.2 the core embedding is not good for small power-law exponents which results in a bad overall embedding. This effect is strong with the temperature 0.5. For higher temperature the quality gets better. The reason for this is that with a higher temperature edges that would be non-edges in the threshold model and non-edges that are edges in the threshold model are more likely. So we have more freedom in the embedding process.

4.4 Phase Dependence

In the following, we investigate whether the associated problems arise in the core phase or the periphery phase. To answer this question we begin with the periphery phase. We run the periphery phase with the core embedding of the core phase and compare it to the results of the periphery phase with the ground truth core embedding and the result of the periphery phase with random core positions. As we can see in Figure 4.3 the algorithm performs better if we use the ground truth of the core embedding and is very bad when using random core positions. We can say that we need a good core embedding in order to get a good overall embedding. It is better to use the estimated core embedding than random core positions but the embedding with estimated core positions is worse than the ground truth embedding. The reason for this is that we do not have an optimal embedding for the core and thus get a worse overall embedding. We investigate the reasons for this in the following section. We can now answer question 3. The core phase is the bottleneck of the algorithm since the overall quality can not be good if the core embedding is bad. We can see that a good core embedding is important for the overall performance of the algorithm.

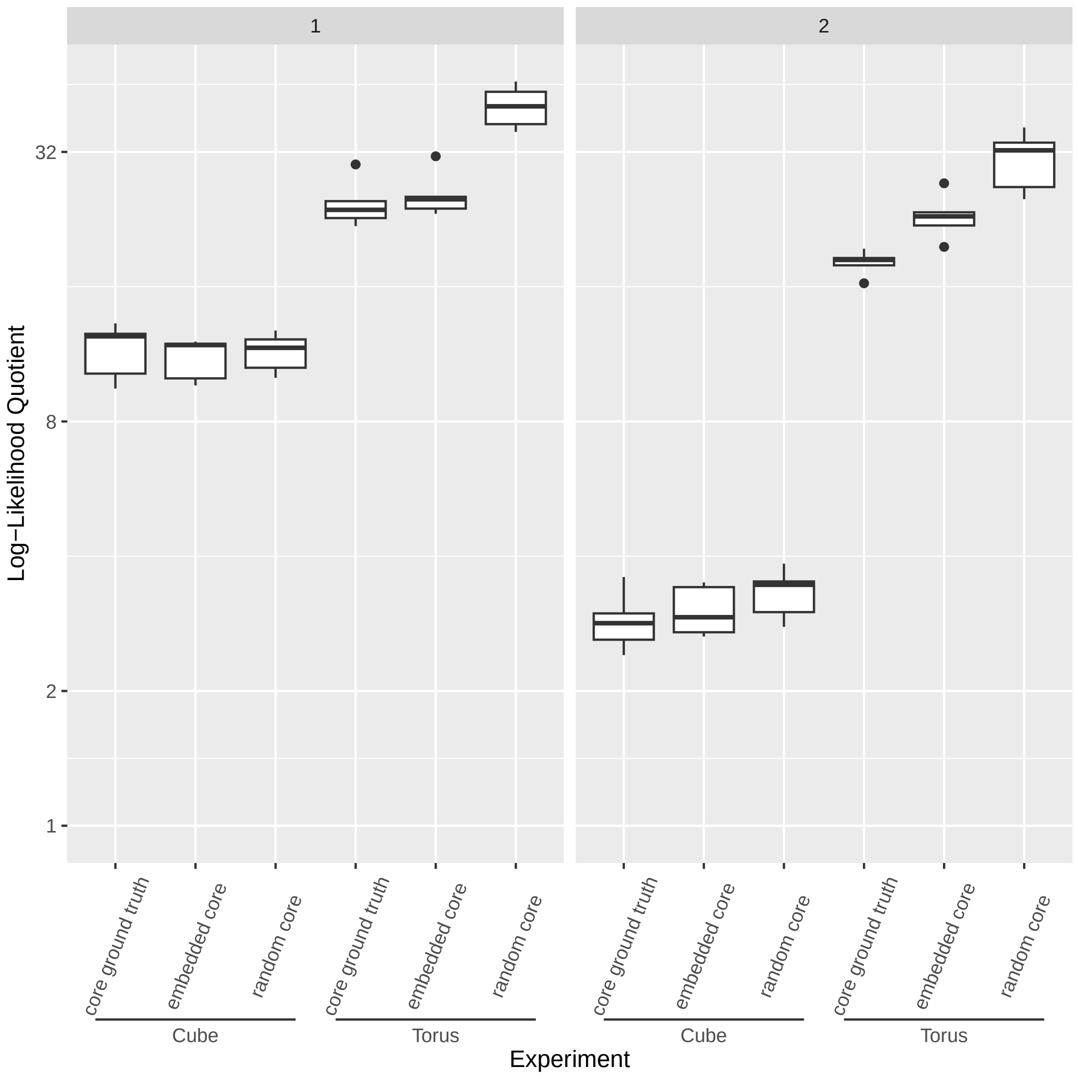


Figure 4.3: Experiment for phase dependence. We generated graphs with 1000 vertices, an average degree of 10, power-law exponent 2.5, temperature 0.1 and a torus and a cube as ground space. We sampled graphs with dimension 1 and 2, where the experiments with dimension 1 are in the left column and the experiments with dimension 2 are in the right column. Then we run the periphery phase three times on the graph with different core embeddings. The first one is the ground truth core embedding, the second embedding is the result of the core phase and the third embedding are random positions for the core vertices.

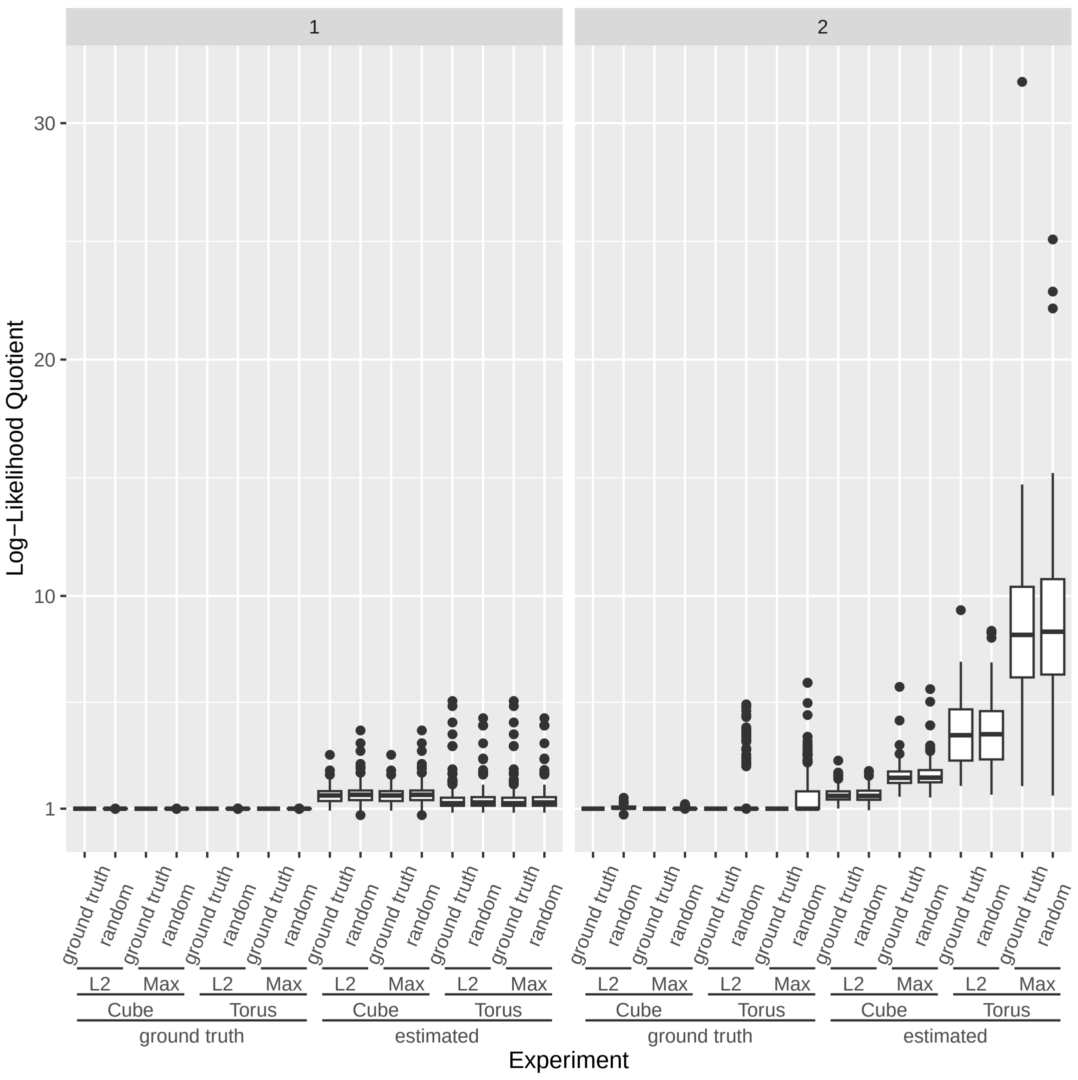
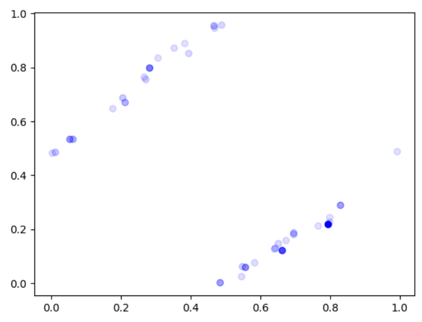
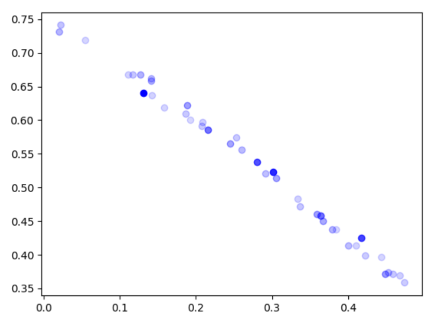


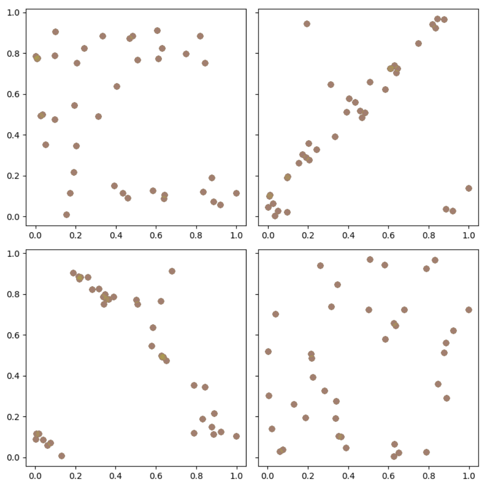
Figure 4.4: Experiment for the core phase quality. We sampled graphs with 20000 vertices, an average degree of 10, power-law exponent 2.5 and temperature 0.1. We sampled graphs with dimension 1 and 2, where the experiments with dimension 1 are in the left column and the experiments with dimension 2 are in the right column. We sampled 200 graphs per experiment configuration. The experiment configuration is given on the x-axis. The used configuration from top to bottom is as follows. We test initialization of the spring embedder with the ground truth and random positions. We used the L_2 and the L_∞ norm. We used a cube and a torus as ground space. We estimated the pairwise distances for core vertices and used the ground truth.



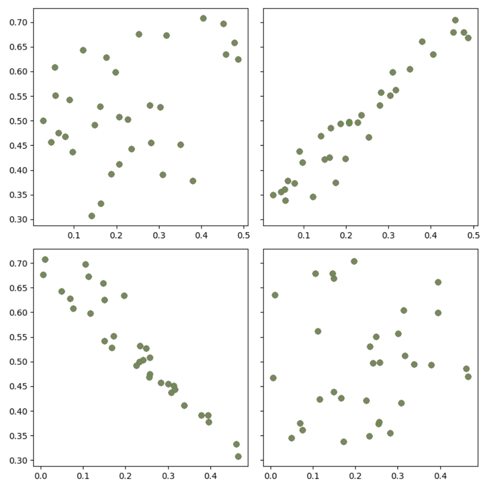
Embedding of the core with dimension 1 and torus as ground space.



Embedding of the core with dimension 1 and cube as ground space.



Embedding of a graph with dimension 2 and torus as ground space.



Embedding of a graph with dimension 2 and cube as ground space.

Figure 4.5: Corresponding-coordinates-plots for the results of the core phase. We sampled a GIRG with 20000 vertices, average degree of 10, power-law exponent 2.5, a temperature of 0.1 which resulted in 33 to 42 core vertices.

4.5 Core Phase Quality

From the previous results we can see that the performance of the algorithm heavily depends on the core embedding. Now we want to see under which parameter configuration the core embedding is bad in order see where the algorithm can be improved in the future. There are several questions we want to answer:

- Does the spring embedder work correctly? Are the given distances reached?
- How good are the estimated distances?
- What is the influence of different norms on the embedding?
- Does the choice of the ground space affect the quality of the core phase?

Therefore, we run the core phase with several parameter combinations. The results can be seen in Figure 4.4. Examples for different ground spaces and dimensions can be seen in Figure 4.5 The first thing we notice is that the quality of the embedding stays the same if we use the ground truth for the initial positions and the ground truth distances between them because in this case the forces are zero. Another thing we can see is that the spring embedder reaches the desired distances because the log-likelihood quotient is close to 1 when we use the ground truth distances and random initial positions. We can see on the basis of dimension 2 that the L_2-norm is always better than the L_∞-norm. When using the L_∞-norm we can only move a vertex in one dimension to get closer to a vertex. In the L_2-norm we use every dimension to change the distance to a vertex. We can answer question 4 with these results. We can see that the estimation is not optimal for higher dimensions because the log-likelihood of the embedding with estimated distances is worse in comparison to embedding with the ground truth distances. When we use a cube as the ground space the quality is better compared to the torus. From these results we can say that the spring embedder works very well whereas the distance estimation especially for higher dimensions can be improved.

4.6 Periphery Phase Quality

In this section we want to evaluate the quality of the periphery phase of the algorithm. Therefore we have several questions:

- How good is the estimation of the initial positions for non-core vertices?
- Does the norm affect the quality of the periphery phase?
- Does the log-likelihood optimization method (gradient ascend vs. sampling) affect the quality of the periphery phase?

The results for this experiment are shown in Figure 4.6. If we focus on the cube we can see that the estimation of the initial position is pretty good because the values for the estimated position and the ground truth are very close. Another thing we notice is that the sampling approach is much better than the gradient ascent. A reason for this could be that we only configured the hyperparameters such as the learning rate for the gradient ascent only for the torus and not for the cube. When looking at the torus the results are not that clear. The estimation of the positions is not that good as with the cube because the quality of the ground truth is always better than when using the estimated positions. The reason for this is that we do not consider the geometry of the torus when we calculate the weighted average. On the torus we need another approach, e.g., we want to minimize the distances to all embedded neighbors for the initial position. The results for the log-likelihood optimization methods differ for dimension 1 and 2. For dimension 1 the sampling approach is better whereas gradient ascent worked better for dimension 2. A reason for this is that we only sample very few positions (7 for n = 1000) which are not enough to find a good position in higher dimensions. We can answer question 5 with these results. With higher dimension the embedding process gets better and the embedding is generally better when using a cube as ground space. We see that this phase also works quite well but the quality there can be improved.

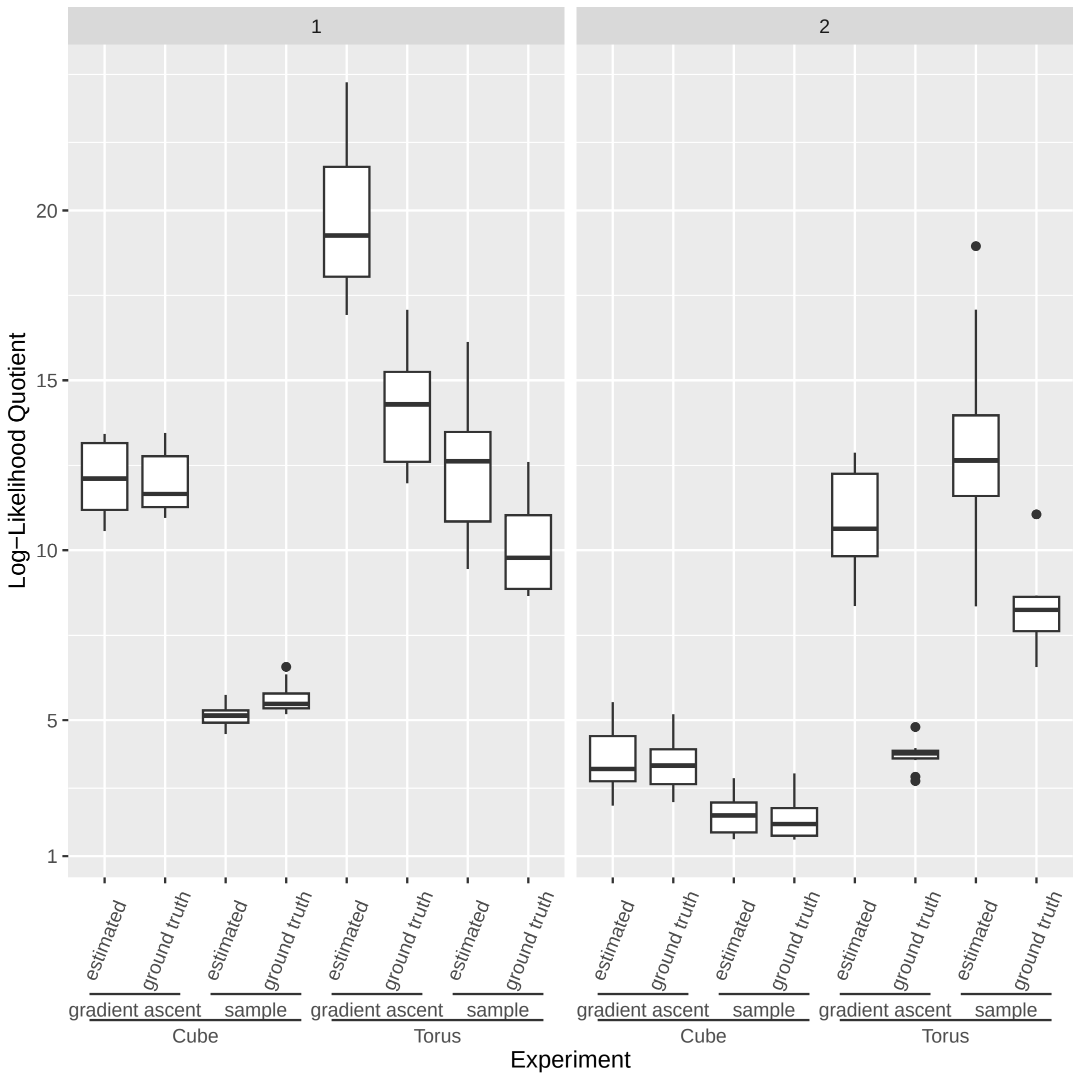


Figure 4.6: Experiment for the periphery phase quality. We generated graphs with 1000 vertices, average degree of 10, power-law exponent 2.5, temperature 0.1 and a torus and a cube as ground space. We used the ground truth embedding for the core and then run the periphery phase. We used different approaches for the log-likelihood method, which are sampling and gradient ascent. We compare the estimation of the initial positions for non-core vertices with the ground truth.

5 Conclusion

We introduced an embedding algorithm for the GIRG model in Chapter 3. The algorithm consists of three steps, the parameter estimation, the core embedding and the periphery embedding. This is the same structure as for the HRG embedder which we used as basis for our algorithm. We applied the techniques of the HRG embedder to the GIRG model and extended them to higher dimensions. We use a spring embedder for the core embedding. The necessary forces for the spring embedder are calculated from the estimated distances between two core vertices. The estimated distance between two vertices is calculated from their common neighborhood. The spring embedder was adapted to the torus and to higher dimensions such that always the shortest distance on the torus is used. After the core is embedded we embed the vertices of the periphery. This is done by calculating the initial position as the weighted average of the embedded neighbors and then iteratively changing the position for each vertex by optimizing its log-likelihood. In the periphery embedding we extended the heuristic approach from the HRG embedder for optimizing the log-likelihood of one vertex. We sampled not just positions inside a sampling radius but we repeat that while decreasing the sample radius. We also introduced a new approach for optimizing the log-likelihood. We use the gradient of the log-likelihood function to maximize the log-likelihood. These are the essential steps for the algorithm.
We also implemented the algorithm and made an extensive evaluation for this algorithm in Chapter 4. We found that the algorithm is working rather well. A good core embedding is important to embed the periphery. An insufficient core embedding leads to a bad overall embedding. Fortunately, the performance of the spring embedder is good and the spring embedder reaches the desired distances. If we look at the periphery phase in an isolated way where we use the ground truth core embedding the resulting embedding is quite good. If we look at graph properties we notice that the quality of the embedding decreases with higher dimensions and increases with higher power-law exponent. If we use a cube instead of a torus as ground space the quality is also better most of the time.
In conclusion, we verified that our algorithm for the GIRG model brings a reduction of complexity compared to the HRG model and extension to higher dimensions as well as a good performance.

5.1 Future Work

Although our algorithm provides good results, there are a few things that can be improved or tested. The improvements can be split in two main areas. The first one is the quality of the embedding and the second one the running time of the algorithm. We saw that the variance for the estimation of the common neighborhood is quite large. This estimation could be improved by providing direct solutions instead of approximations or by using other topological features of the graph to calculate the common neighborhood respectively the distance between two vertices. In the periphery phase we could improve the estimation of the initial position. Currently the position is calculated as if we would use a cube. For the torus it is not clear what the weighted average is. A promising approach is to choose the initial

position such that the distance to all other neighbors is minimized. In the periphery phase we could also improve the gradient ascent. There we can add a second hyperparameter to decrease the learning rate. Another idea is to decrease the learning rate for all vertices that are already embedded. It is more likely that they already have a good position which is lost when the learning rate is too large. It would be also helpful to find other methods to measure the quality of the embedding such as quantifying the corresponding-coordinates-plots with linear regression. Additionally we would like to compare our embedder to the hyperbolic embedder.
Regarding the running time we need to speed up the log-likelihood calculation. To perform the speedup as described in this thesis we need a geometric data structure to represent the graph. The data structure should support higher dimensions and it should be a dynamic data structure. The data structure of the hyperbolic embedder can not be used because of the one dimension. Another data structure for GIRGs in [Blä+22] also does not fit the requirements because the position of the vertices can not be dynamically changed. So we need to find a data structure that meets both requirements.
Furthermore, we see a good opportunity to perform challenging tests of our algorithm when applying it to real world data and and expect promising results.

Bibliography

[BF22] Thomas Bläsius and Philipp Fischbeck. "On the External Validity of Average-Case Analyses of Graph Algorithms". In: *30th Annual European Symposium on Algorithms, ESA 2022, September 5-9, 2022, Berlin/Potsdam, Germany.* Edited by Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman. Vol. 244. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 21:1–21:14. DOI: *10.4230/LIPIcs.ESA.2022.21.*

[BFK21] Thomas Bläsius, Tobias Friedrich, and Maximilian Katzmann. "Force-Directed Embedding of Scale-Free Networks in the Hyperbolic Plane". In: *The Sea.* 2021.

[BFKL16] Thomas Bläsius, Tobias Friedrich, Anton Krohmer, and Sören Laue. "Efficient Embedding of Scale-Free Graphs in the Hyperbolic Plane". In: *24th Annual European Symposium on Algorithms (ESA 2016).* 2016, 16:1–16:18. DOI: *10.4230/LIPIcs.ESA.2016.16.*

[Blä+22] Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. "Efficiently generating geometric inhomogeneous and hyperbolic random graphs". In: *Netw. Sci.* Volume 10 (2022), pp. 361–380. DOI: *10.1017/nws.2022.32.*

[BS03] Stefan Bornholdt and Heinz Georg Schuster, eds. *Handbook of graphs and networks : from the genome to the internet.* Weinheim, 2003.

[CSN07] Aaron Clauset, Cosma Rohilla Shalizi, and Mark E. J. Newman. "Power-Law Distributions in Empirical Data". In: *SIAM Rev.* Volume 51 (2007), pp. 661–703.

[FGKS23a] Tobias Friedrich, Andreas Göbel, Maximilian Katzmann, and Leon Schiller. *A simple statistic for determining the dimensionality of complex networks.* 2023. arXiv: *2302.06357.*

[FGKS23b] Tobias Friedrich, Andreas Göbel, Maximilian Katzmann, and Leon Schiller. "Cliques in High-Dimensional Geometric Inhomogeneous Random Graphs". In: *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, July 10-14, 2023, Paderborn, Germany.* Edited by Kousha Etessami, Uriel Feige, and Gabriele Puppis. Vol. 261. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 62:1–62:13. DOI: *10.4230/LIPIcs.ICALP.2023.62.*

[GF18] Palash Goyal and Emilio Ferrara. "Graph embedding techniques, applications, and performance: A survey". In: *Knowl. Based Syst.* Volume 151 (2018), pp. 78–94. DOI: *10.1016/j.knosys.2018.03.022.*

[Keu18] Ralph Keusch. "Geometric Inhomogeneous Random Graphs and Graph Coloring Games". en. Zurich: ETH Zurich, 2018. DOI: *10.3929/ethz-b-000269658.*

[Kob13] Stephen G. Kobourov. "Force-Directed Drawing Algorithms". In: *Handbook of Graph Drawing and Visualization.* 2013.

[Kri+10] Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. “Hyperbolic geometry of complex networks”. In: *Phys. Rev. E* Volume 82 (Sept. 2010), p. 036106. DOI: *10.1103/PhysRevE.82.036106*.

[LK07] David Liben-Nowell and Jon M. Kleinberg. “The link-prediction problem for social networks”. In: *J. Assoc. Inf. Sci. Technol.* Volume 58 (2007), pp. 1019–1031. DOI: *10.1002/asi.20591*.

[New01] M. E. J. Newman. “Clustering and preferential attachment in growing networks”. In: *Phys. Rev. E* Volume 64 (July 2001), p. 025102. DOI: *10.1103/PhysRevE.64.025102*.

[PPK15] Fragkiskos Papadopoulos, Constantinos Psomas, and Dmitri V. Krioukov. “Network Mapping by Replaying Hyperbolic Growth”. In: *IEEE/ACM Trans. Netw.* Volume 23 (2015), pp. 198–211. DOI: *10.1109/TNET.2013.2294052*.

[Tan+15] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. “LINE: Large-Scale Information Network Embedding”. In: *Proceedings of the 24th International Conference on World Wide Web*. Florence, Italy: International World Wide Web Conferences Steering Committee, 2015, pp. 1067–1077. ISBN: 9781450334693. DOI: *10.1145/2736277.2741093*.

[VHHK19] Ivan Voitalov, Pim van der Hoorn, Remco van der Hofstad, and Dmitri Krioukov. “Scale-free networks well done”. In: *Phys. Rev. Res.* Volume 1 (Oct. 2019), p. 033034. DOI: *10.1103/PhysRevResearch.1.033034*.

[WHWW21] Liping Wang, Fenyu Hu, Shu Wu, and Liang Wang. “Fully Hyperbolic Graph Convolution Network for Recommendation”. In: *CIKM ’21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*. Edited by Gianluca Demartini, Guido Zuccon, J. Shane Culpepper, Zi Huang, and Hanghang Tong. ACM, 2021, pp. 3483–3487. DOI: *10.1145/3459637.3482109*.