# Efficient Embedding of Scale-Free Graphs in the Hyperbolic Plane

**Thomas Bläsius[1], Tobias Friedrich[2], Anton Krohmer[3], and Sören Laue[*4]**

1 **Hasso Plattner Institute, Potsdam, Germany**
`thomas.blaesius@hpi.de`
2 **Hasso Plattner Institute, Potsdam, Germany**
`tobias.friedrich@hpi.de`
3 **Hasso Plattner Institute, Potsdam, Germany**
`anton.krohmer@hpi.de`
4 **Friedrich Schiller University, Jena, Germany**
`soeren.laue@uni-jena.de`

## Abstract

Hyperbolic geo metry appears to be intrinsic in many large real networks. We construct and implement a new maximum likelihood estimation algorithm that embeds scale-free graphs in the hyperbolic space. All previous approaches of similar embedding algorithms require a runtime of $\Omega(n^2)$. Our algorithm achieves quasilinear runtime, which makes it the first algorithm that can embed networks with hundreds of thousands of nodes in less than one hour. We demonstrate the performance of our algorithm on artificial and real networks. In all typical metrics like Log-likelihood and greedy routing our algorithm discovers embeddings that are very close to the ground truth.

**1998 ACM Subject Classification** F.2.2 Nonnumerical Algorithms and Problems

**Keywords and phrases** hyperbolic random graphs, embedding, power-law graphs, hyperbolic plane

**Digital Object Identifier** 10.4230/LIPIcs.ESA.2016.16

## 1 Introduction

The study and analysis of complex real-world networks is a rapidly growing field. There are a number of commonly observed properties of complex networks like power-law degree distribution, small clustering coefficient, and small average distances. During the last decade, dozens of models for such *scale-free networks* have been proposed. The most popular model is the preferential attachment model by Barabási and Albert [5]. Most accessible for mathematical analysis is the inhomogeneous random graph model by van der Hofstad [33], which generalizes the models of Chung and Lu [10, 1, 2] and Norros and Reittu [26].

All aforementioned network models observe a power-law degree distribution, small diameter and average distances. However, all of them naturally also have a *small clustering coefficient*, that is, the number of triangles and small cliques in such artificial networks is magnitudes lower than observed in real-world networks. The reason is that in the standard definitions of these network models, the edges are (merely) independent, which is not true

* Sören Laue acknowledges the support of Deutsche Forschungsgemeinschaft (DFG) under grant GI-711/5-1 within the priority program "Algorithms for Big Data".




for real-world networks. For social networks the reason is easy to see: If someone is friends with two people, it is likelier that they know each other as well than it would be for two random strangers to forge a connection. There are a number of modifications of above models that incorporate this intuition [34, 25, 23], however, all these fixes introduce other artificial artifacts and can not explain *why* the clustering occurs in the first place.

**Hyperbolic Random Graphs.** A natural definition of a scale-free network model with all aforementioned properties emerges when adding an appropriate geometry. It is well studied that geometric random graphs with an Euclidean space result in a Poisson degree distribution [30]. Krioukov et al. [20] took a different approach by assuming an underlying *hyperbolic geometry* to the network. The most prominent feature of a hyperbolic space is its exponential expansion around a given point, in contrast to Euclidean space, which expands only polynomially. *Hyperbolic random graphs* are obtained by placing all nodes in the hyperbolic plane, and connecting two nodes whenever they are a small (hyperbolic) distance apart. The desired clustering then naturally emerges as a reflection of the geometric proximity. This model has been analyzed to have a power-law degree distribution and high clustering [16, 20], to have a polylogarithmic diameter and ultra-short average distances of order $\mathcal{O}(\log\log n)$ [15, 9], and allows fast bootstrap percolation [19].

**Generating Hyperbolic Random Graphs.** With most fundamental structural properties of hyperbolic random graphs settled, the next step is studying algorithms on the network model. The first addressed algorithmic problem is efficiently *generating* such a graph or, equivalently, *sampling* a graph from the probability distribution defined by hyperbolic random graphs. The naive generation of a hyperbolic random graph takes $\Theta(n^2)$ time [3]. Using a polar quadtree adapted to hyperbolic space, von Looz et al. [36] achieved a time complexity of $\mathcal{O}((n^{3/2} + m)\log n)$; and by a more sophisticated partitioning of the space, Bringmann et al. [9] obtained an optimal expected linear runtime for generation, which is crucial for large-scale experiments.

**Visualizing Data in Hyperbolic Geometry.** It is well known in the visualization community that hierarchical or tree-like structures can be well represented in a hyperbolic space [32]. There are three approaches to embed a network in the hyperbolic space:

- A popular way to obtain hyperbolic coordinates for the nodes of a network is embedding a spanning tree of the network in hyperbolic space [38, 37, 24]. As trees can be embedded perfectly, this is a very efficient way to map a network and has been used for interactive network browsers, which allow assigning more display space to the interesting portions of a network [21, 22]. The result might reduce visual clutter and help focus, but it ignores most structural details of the network. Nodes which are close in graph distance are not necessarily close in hyperbolic space. In fact, clusters and most local structures are not preserved.
- Another approach is determining shortest path distances and finding an embedding where metric distances match the graph distances. Computing the all-pair-shortest-path matrix can be done with the well established Euclidean data analysis method Multidimensional Scaling (MDS) [13], which has been translated to hyperbolic geometry [12]. Due to the quadratic size of the distance matrix, this approach only works for graphs with a few hundred nodes [4]. To reduce the runtime, it is possible to (randomly) select a small subset of the pairwise distances [31, 35, 42].

- Our objective is slightly different. Instead of preserving distances between nodes, we aim at inferring the *popularity* (reflected by radial coordinates) and *similarity* (reflected by angular coordinates) of all nodes [28]. The reason why a connection between vertices exist can be twofold: Either, the two vertices are similar, which holds e. g. for close friends in social networks; or for geographically close ASs in the Internet graph. On the other hand, a connection may be present due to the popularity of one end vertex: For instance, many people follow Lady Gaga on Twitter; but most are arguably not very similar to her. Embedded shortest path distances lose this information. Our goal is to recover this information using the most likely embedding assuming a hyperbolic nature of the graph in the first place. For this, we use the random network model of Krioukov et al. [20].

**Maximum Likelihood Estimation Embedding of Graphs in Hyperbolic Space.** We focus on the last-mentioned approach of maximum likelihood estimation (MLE) algorithms, i.e., we want to find the node coordinates in the network by maximizing the probability that the network is produced by some underlying hyperbolic model. Boguñá et al. [8] were the first to find such an embedding for the Internet graph ($m = 58\,416$ connections between $n = 23\,752$ autonomous systems) in the hyperbolic space. It is impressive that greedy navigation along these hyperbolic coordinates is almost maximally efficient, i.e., it almost always finds the shortest paths between almost any two pairs of vertices in the same component. However, the described method to discover the hyperbolic coordinates "require[s] substantial manual intervention and do[es] not scale to large networks" [20]. A general algorithm for embedding a network in a hyperbolic space was later presented by Papadopoulos et al. [29]. Their HyperMap algorithm is an approximate maximum likelihood estimation (MLE) algorithm. They demonstrate their algorithm on synthetic networks with $n = 5\,000$ nodes and $m = 20\,000$ edges and a subset of the aforementioned Internet graph with $n = 8\,220$ nodes. The asymptotic runtime was improved in a subsequent paper from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$ [27]. The authors present no runtime measurements [29, 27], but their HyperMap code on our machine requires more than 1.5 hours for a graph of size 2 000 (cf. Section 6.2). The algorithm was further refined in [39], who use a community detection algorithm for the coarse layout of the nodes; and an MLE to find precise positions. While their runtime is still $\Omega(n^2)$, our techniques extend to their case.

**Our New Hyperbolic Embedder.** We design and implement a new algorithm for computing hyperbolic MLE embeddings of massive networks (Section 5).[1] Compared to previous approaches that need $\Omega(n^2)$ runtime, our algorithm runs in quasilinear runtime. To this end, we developed several new techniques. First, we use an analytical approach to compute the expected angles between pairs of high-degree nodes based on their number of common neighbors. In contrast to [27], this approach does not rely on expensive numerical computations, making it fast in practice. The resulting angle distance matrix is then fed to a spring embedder that finds good positions for high-degree nodes in linear time. For small degree nodes, we substantially improve runtime by using the geometric data structure of Bringmann et al. [9] that allows traversing nodes of close proximity in expected amortized constant time.

This enables us to embed significantly larger graphs than before. For instance, we computed in under one hour a hyperbolic embedding of the Amazon product recommendation network which has over 300 000 nodes. To evaluate the quality of our embedding, we conduct large-scale experiments on 6 250 generated graphs and compare our embedding with the

[1] Our code will be made available at `https://hpi.de/friedrich/research/hyperbolic`.

ground truth data (Section 6). We observe that in typical metrics like Log-likelihood and greedy routing, our algorithm achieves embeddings that are competitive with the original.

Furthermore, we investigate the performance of two classical methods of embedding graphs in the Euclidean space, namely spring embedders and maximum variance unfolding, when applied to the hyperbolic space (Sections 3 and 4). We find that both of them can work under some strong assumptions, but generally fail to translate to large real-world graphs.

## 2 Preliminaries

In this section, we briefly introduce the hyperbolic random graph model. Due to space constraints, we keep the definitions concise and refer the reader to previous work for a more intuitive introduction, see e.g. [20, 16]. We use the native representation of the hyperbolic space [20] of curvature $-1$, where points are identified by radial coordinates $(r, \varphi)$. The first coordinate describes the hyperbolic distance from the origin, and two points $x, y$ have hyperbolic distance

$$\operatorname{dist}(x, y) := \cosh^{-1}(\cosh(r_x)\cosh(r_y) - \sinh(r_x)\sinh(r_y)\cos(\varphi_x - \varphi_y)).$$

The hyperbolic random graph model formally defines a probability distribution over the set of all graphs of size $n$. A graph $G$ on $n$ vertices is sampled from this distribution as follows. Consider a disc $D_n$ of radius $R = 2\log n + C$ in the hyperbolic space, where $C$ is a parameter adjusting the average degree of the resulting graph. Each vertex $v$ is randomly equipped with hyperbolic coordinates $(r_v, \varphi_v)$ sampled from the probability density function $f(r, \varphi) = \frac{\alpha \sinh(\alpha r)}{2\pi(\cosh(\alpha R)-1)}$, for a parameter $\alpha$ adjusting the power-law exponent $\beta = 2\alpha + 1$ of the resulting network. Then, every two vertices $u, v$ are connected with probability

$$p(\operatorname{dist}(u, v)) := \left(1 + \exp\left(\tfrac{1}{2T} \cdot (\operatorname{dist}(u, v) - R)\right)\right)^{-1}, \tag{2.1}$$

where $T$ is a parameter regulating the importance of the underlying geometry: When $T \to 0$, we obtain the so-called *step model*, where an edge $\{u, v\}$ is present if and only if $\operatorname{dist}(u, v) \leqslant R$. For $T > 0$, we obtain the *binomial model*, where long-range edges are possible (but unlikely). Typically, one assumes $0 \leqslant T < 1$. This yields a random graph depending on 4 parameters: $n, R$ (or $C$), $\alpha$, and $T$. Following standard graph notation, we write $\Gamma(v)$ for the set of neighbors of $v$, and we use $\delta$ to refer to the average degree of $G$.

Further, given a graph $G = (V, E)$ and any mapping from nodes to hyperbolic coordinates $\{r_i, \varphi_i\}_{i=1}^n$, we define the *Log-likelihood* as

$$\mathcal{L}(\{r_i, \varphi_i\}_{i=1}^n \mid G) := \sum_{\{u,v\} \in E} \log(p(\operatorname{dist}(u, v))) + \sum_{\{u,v\} \notin E} \log(1 - p(\operatorname{dist}(u, v))),$$

where the hyperbolic distances dist are taken with respect to the coordinates $\{r_i, \varphi_i\}_{i=1}^n$. To simplify presentation, we write

$$\mathcal{L}(v) := \sum_{u \in \Gamma(v)} \log(p(\operatorname{dist}(u, v))) + \sum_{u \notin \Gamma(v)} \log(1 - p(\operatorname{dist}(u, v))), \tag{2.2}$$

so that we have $\mathcal{L}(\{r_i, \varphi_i\}_{i=1}^n \mid G) = \frac{1}{2}\sum_{v \in V} \mathcal{L}(v)$.

Our goal is to devise an algorithm which, given only the network structure (i.e. a list of edges) of a generated hyperbolic random graph, can re-infer the hyperbolic coordinates of the original embedding. As additional requirements, we would like that the algorithm is robust to noise (i.e. works reasonably well even if the supplied graph was not hyperbolic).

Before presenting our algorithm, we revisit two popular embedding techniques in the Euclidean plane and investigate their performance when applied to the hyperbolic setting.

# 3 Spring Embedder

A heavily used technique to embed graphs in the Euclidean plane is the force-directed method (also called spring embedder) [17], which works roughly as follows. For every edge one assumes an attractive force pulling its end vertices toward each other, and for every pair of vertices one assumes a repulsive force pushing them away. The algorithm starts with some initial drawing (e.g., by choosing random positions) and computes for each vertex the total force acting on it. Then, all vertices are moved by a small step according to these forces. This is iterated until a stable configuration is reached.

In a drawing generated by a spring embedder, edges are usually short and non-adjacent vertices are usually far away from each other. Moreover, the repulsive forces lead to a somewhat uniform distribution of the vertices in the available space. Note that these are exactly the properties we wish to obtain for our embeddings in the hyperbolic plane. It thus seems natural to adapt spring embedders to the hyperbolic geometry, which actually has been done before by Kobourov and Wampler [18]. In the following we discuss why the straight-forward way of implementing a spring embedder in the hyperbolic plane does not work in our setting. For several adaptations that lead to good results at least for smaller graphs, see the online version.

## 3.1 Difficulties in the Hyperbolic Plane

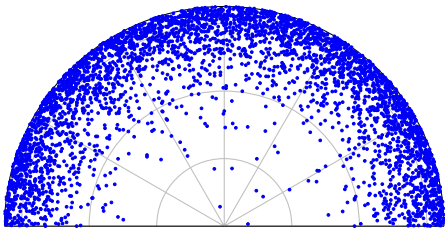
To understand the difficulties in the hyperbolic plane, first consider the following artificial situation in the Euclidean plane. Assume $v$ is a vertex only connected to $u$; and assume the current drawing is already stable except that $v$ is far away from $u$. Now when $v$ moves towards $u$, it also gets closer to other vertices it is not connected to, which then push $v$ back towards the direction where it came from. This is not a problem, however, as there are usually only few vertices close enough to $v$ such that their force is noticeable. Moreover, vertices on the opposite side of $v$ support the movement towards $u$.

In the hyperbolic plane, an analogous situation works out differently. The geodesic line between $v$ and $u$ contains points with smaller radius, such that $v$ first moves almost directly towards the origin. In turn, the distance to *all* other nodes decreases, which immediately pushes $v$ back to a position with larger radius. Thus, even bad embeddings are stable.

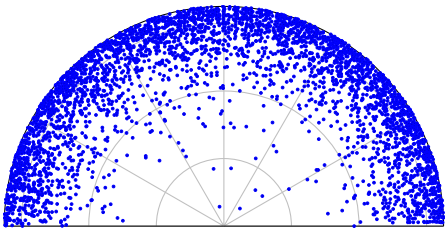
Judging from the pictures presented by Kobourov and Wampler [18], it seems that they did not encounter these issues in their spring embedder. This can be explained by the fact that the radii they use are all rather small, which can be deduced from the presented drawings by observing that the vertices are very well separated from the boundary of the Poincaré disk (which is only true for very small radii). However, for such small radii the hyperbolic plane behaves very similar to the Euclidean plane. We note that using small radii is reasonable for visualizing small graphs using a fish-eye view. However, as the radii in a hyperbolic random graph grow logarithmically with an increasing number of vertices, this is not suitable for our purpose.

# 4 Maximum Variance Unfolding

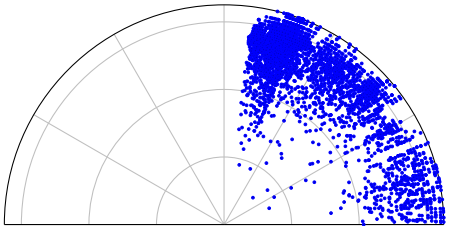
Another popular method for embedding graphs into the Euclidean plane is maximum variance unfolding (MVU) [40]. This is essentially a semidefinite program whose objective function spreads out nodes while using constraints to keep neighbors close together. In the one-dimensional case it is equivalent to an LP.



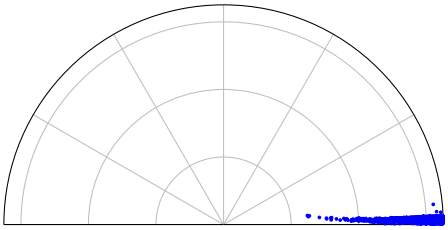
**(a)** Original Points (edges not shown) of a hyperbolic random with $T = 0$.



**(b)** Embedded nodes using the LP. All parameters except the angular coordinates were given as additional information. The embedding is almost equivalent to the original.



**(c)** Embedded nodes using the LP with estimated radial coordinates (See Section 5.1). The quality of the LP solution quickly degrades.



**(d)** Embedded nodes using the LP with all other parameters given. The graph was generated using $T = 0.5$. The embedding is essentially unusable.

**Figure 1** First phase of the LP. Since nodes are placed in $[0, \pi]$, half of $D_n$ is hidden.

The use-case in the hyperbolic geometry is similar: Nodes shall have distance $< R$ if they have an edge, and distance $\geqslant R$ otherwise. It is possible to encode this into the following LP:

$$
\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{n} \varphi_j \\
\text{subject to} \quad & \varphi_i - \varphi_j \leqslant \theta(r_i, r_j), && i, j = 1, \dots, n, \quad i \neq j \\
& \varphi_j - \varphi_i \leqslant \theta(r_i, r_j), && i, j = 1, \dots, n, \quad i \neq j \\
& 0 \leqslant \varphi_i \leqslant \pi && i = 1, \dots, n \\
& \varphi_v = 0, && \text{for some starting node } v
\end{aligned}
$$

where $\theta(r_i, r_j)$ is the maximal angular distance such that nodes $\operatorname{dist}(i, j) \leqslant R$, i. e.

$$\theta(r_i, r_j) = \arccos\left(\frac{\cosh(r_i)\cosh(r_j) - \cosh(R)}{\sinh(r_i)\sinh(r_j)}\right). \tag{4.1}$$

The LP has a caveat: It is only able to spread nodes on the half circle $[0, \pi]$; since for larger angular coordinates the hyperbolic distances start decreasing again, which is not encodable in the LP. This can be fixed, however, using a small trick: First, embed all nodes on a half-circle with an arbitrary starting node $v$. Then, pick the node $u$ in the embedding with angular coordinate closest to $\frac{\pi}{2}$; and embed the graph again using $u$ as the starting node.

This yields all nodes that belong in the lower half of $D_n$: If $w$ has angular distance at least $\frac{\pi}{2}$ from $u$ in the second embedding, we set $\varphi_w = \varphi_w + \pi$ in the first embedding.

This simple method works surprisingly well on generated hyperbolic random graphs that are drawn from the step model, when given all global parameters and radial coordinates, see Figure 1a–b. It is, however, extremely volatile to the quality of the estimated parameters; and it fails completely when used on a real graph or even a graph generated by the binomial

**Algorithm 1** Fast Embedding Algorithm

**Input:** Undirected connected Graph $G = (V, E)$

1: Estimate global parameters $n, R, \alpha, T$; and radial coordinates $r_i$ ▷ See Section 5.1
2: Partition nodes into layers such that $v \in L_i \Leftrightarrow \deg(v) \in [2^i, 2^{i+1} - 1]$
3: Embed Core (all nodes in layers $\geqslant \frac{\log n}{2}$) ▷ See Section 5.2
4: **for** $i = \frac{\log n}{2} - 1 \dots 0$ **do**
5: **for** $r = 1 \dots \log n$ **do**
6: **for all** $v \in \bigcup_{j \geqslant i} L_j$ **do**
7: Embed $v$ by optimizing its Log-likelihood ▷ See Sections 5.3 and 5.4

model, see Figure 1c–d. The reason is that the LP has a constraint for each edge in the graph: If there is just one long-range edge, the MVU can no longer unfold the graph and all nodes are mapped to an extremely small range of angular coordinates. This behavior persists even after adding different error terms for edges; and we were not able to make this approach work on noisy data.

# 5 The Embedder

Our embedding algorithm is inspired by the Metropolis-Hastings Algorithm from [8]. Algorithm 1 contains a bird's eye view over the whole algorithm. Detailed description of individual steps follow in the next sections.

The algorithm proceeds in three phases: First, it estimates all parameters that are computationally easy to guess. This includes the radial coordinates of all nodes, see Section 5.1.

In the second phase, high-degree nodes are embedded by considering their common neighbors. Producing a good initial ordering of nodes in inner layers is crucial for the success of the algorithm since nodes in all subsequent layers are typically placed close to their neighbors in higher layers. This step is described in Section 5.2.

In the third phase, the algorithm embeds the rest of the graph layer-wise. To embed a layer $L_i$, we iterate over all nodes $v \in L_i$. In each iteration, $\mathcal{O}(\log n)$ angular coordinates for $v$ are sampled; and $v$ is moved to the position with the best Log-likelihood, see Sections 5.3 and 5.4. This is repeated $\log n$ times per layer. While this step is similar to HyperMap [8, 27, 29], we improve upon their algorithm by achieving an amortized polylogarithmic runtime per node as compared to their linear runtime. Our overall algorithm thus runs in $\mathcal{O}(n \cdot \text{polylog}(n))$.

## 5.1 Parameter Estimation

To bootstrap the embedding algorithm, the global graph parameters have to be known: The original number of nodes $n$, the radius $R$ of the disc $D_n$, the parameter $\alpha$ adjusting the power-law exponent; and the parameter $T$ adjusting the clustering. These values are required for instance for evaluating the probability that two nodes are connected, see equation (2.1) which in turn is needed to produce the Log-likelihood. In the following, we give some brief explanations on how each parameter is guessed.

**Estimating $n$.** Algorithm 1 expects a connected graph as input, since disconnected components can be placed anywhere in the graph as there is no adjacency information.

The hyperbolic random graph, however, does typically not produce a connected graph. For power-law exponents $2 < \beta < 3$, its giant component is of size $\Theta(n)$ [6, 7]; and for $\beta \geqslant 3$ the graph breaks up into components of order $o(n)$. Unfortunately, the leading constant of

the size of the giant component is yet unknown; and a numerical estimation is hard since it is governed by a non-linear system of equations together with other parameters [8].

We have found experimentally that the majority of nodes missing from the giant component are of degree 0. Surprisingly, the most effective and robust method for estimating the number of these nodes was by simply extrapolating from the number of 1- and 2-degree nodes. Let $\hat{n} \cdot F(k)$ be the number of nodes of degree $k$, where $\hat{n}$ is the total number of nodes in the input graph. Then, we estimate $n$ simply by setting $n := \hat{n}(1 + \max\{0, 2F(1) - F(2)\})$.

**Estimating $\alpha$.** The parameter $\alpha$ adjusts the power-law exponent $\beta$ of the hyperbolic random graph via the functional behavior $\beta = 2\alpha + 1$ [16]. We estimate $\beta$ from the cumulative degree distribution using the classical algorithm by Clauset et al. [11].

**Estimating $T$.** Recall that this parameter adjusts the importance of the underlying geometric structure. It has recently been observed, however, that $T$ does not have a big influence on the quality of the embedding [27]. We found that setting $T$ to a small fixed value like 0.1 produces good results. We investigate the role of $T$ closer in the online version.

**Estimating $R$ and $r_i$.** We estimate these values using the above determined parameters. Good analytical estimates have been derived in previous work [8]:

$$R = 2\log\left(\frac{4n^2\alpha^2 T}{|E| \cdot \sin(\pi T)(2\alpha - 1)^2}\right), \quad r_i = \min\left\{R, \quad 2\log\left(\frac{2n\alpha T}{\deg(i) \cdot \sin(\pi T)(\alpha - \frac{1}{2})}\right)\right\}$$

## 5.2 Embedding the Core

Laying out the large-degree nodes (also called the *core* of the graph) has a huge impact on the overall performance of the embedding. We consider all nodes $v$ with radial coordinates $r_v < R/2$ to be in the core, of which there are $\Theta(n^{1-\alpha})$ in expectation [14]. If the node ordering of the core is roughly correct, the algorithm will usually yield excellent embeddings. One the other hand, if the core was embedded poorly, the remaining steps can not salvage the poor initialization. Thus, we put considerable care into embedding the core correctly.

HyperMap [29] uses the number of common neighbors of large degree nodes to lay out the core: For two nodes $u, v$ they compute the number $c_{uv} = |\Gamma(u) \cap \Gamma(v)|$, and numerically determined the angle $\varphi(c_{uv}, r_u, r_v)$ that maximizes the likelihood that the nodes $u, v$ have $c_{uv}$ common neighbors. This is a promising approach, as the common neighborhood of large nodes is tightly concentrated around its expected value. Determining the likelihood numerically, however, is a computationally expensive operation.

To overcome this, we analytically derive in Section 5.2.1 an approximate expression for the relative angle of two nodes up to constant factors. Using this, we present a spring embedder in Section 5.2.2 that embeds the core based on the estimated pair-wise angle differences.

### 5.2.1 Estimating the Angle-Differences

To estimate the relative angle between two nodes, we use their inferred radial coordinates and the number of their common neighbors. We perform this computation in the step model; however, we have experimentally found that our results hold up well in the binomial model.

Let $u, v$ be the two nodes whose (expected number of) common neighbors we wish to compute. They have radii $r_u$ and $r_v$, respectively, and a relative angle of $\Delta\theta(u, v)$. W. l. o. g., we assume that $r_u \leqslant r_v$. Consider now a third node $w$. We compute the probability that $w$

is connected to both $u$ and $v$. Under the assumption that $r_u + r_w \geqslant R$ and $r_v + r_w \geqslant R$, we know from [16] that this only holds if

$$\begin{aligned}\Delta\theta(u,w) &\leqslant 2e^{\frac{1}{2}(R-r_u-r_w)}(1+\Theta(e^{R-r_u-r_w})), \quad \text{and}\\ \Delta\theta(v,w) &\leqslant 2e^{\frac{1}{2}(R-r_v-r_w)}(1+\Theta(e^{R-r_v-r_w})).\end{aligned} \tag{5.1}$$

Assume $r_v + r_w \geqslant R$ does not hold. In this case, the distance between $v$ and $w$ is obviously at most $R$ and thus they are connected. Moreover, note that in this case the right hand side of the above formula increases with increasing $R$ and thus the inequality is satisfied for any angle $\Delta\theta(v,w)$ if $R$ is sufficiently large. Thus, under the assumption that $R$ is sufficiently large, we may use equation (5.1).

Observe now that for large enough radii $r_w$, the node $w$ is not connected to either $u$ or $v$ (unless $\Delta\theta(u,v) \leqslant \mathcal{O}(\frac{1}{n})$). On the other hand, when $R - r_v - r_w = \Omega(1)$, $w$ is connected with constant probability to both $u$ and $v$. Thus, depending on the radius $r_w$, there is a "good" fraction of the angular coordinates $[0, 2\pi)$ where $w$ will be connected to both nodes, and a "bad" fraction where it will be connected to only one or neither of $u, v$. We call the probability to be connected to both nodes $p_g(r_w)$.

We already know that $p_g(r_w) = 1 \Leftrightarrow r_w = R - r_v \pm \Theta(1)$. We label this critical value of $r_w$ with $r_1$. On the other hand, $p_g(r_w) = 0$ holds when $\theta(r_u, r_w) + \theta(r_v, r_w) \leqslant \Delta\theta(u,v)$, since then there is no possible angle for $\varphi_w$ where it is connected to both nodes $u, v$, see equation (4.1). The critical value $r_0$ for which this number becomes positive is when $\theta(r_u, r_w) + \theta(r_v, r_w) = \Delta\theta(u,v)$ and thereby

$$\begin{aligned}\Delta\theta(u,v) &= 2e^{\frac{1}{2}(R-r_u-r_0)}(1\pm\Theta(e^{R-r_u-r_0})) + 2e^{\frac{1}{2}(R-r_v-r_0)}(1\pm\Theta(e^{R-r_v-r_0}))\\ &= \Theta(1)\cdot e^{\frac{1}{2}(R-r_u-r_0)}.\end{aligned}$$

Solving for $r_0$, this holds whenever $r_0 = \min\{R, R - r_u - 2\log(\Delta\theta(u,v)) \pm \Theta(1)\}$.

For values $r_1 \leqslant r_w \leqslant r_0$, the regions in which $w$ connects to $u, v$ both increase as in equation (5.1). Thus, the intersection of these regions increases as $p_g(r_w) \sim e^{-r_w/2}$. To determine the function up to constants, we set

$$1 = p_g(r_1) = A\cdot e^{-r_1/2} + B, \quad \text{and} \quad 0 = p_g(r_0) = A\cdot e^{-r_0/2} + B.$$

Solving this system of equations, we obtain that $p_g(r_w) = \Theta(1)\cdot(e^{\frac{1}{2}(r_1-r_w)} - e^{\frac{1}{2}(r_1-r_0)})$. Thus, we may compute the probability that an arbitrary node is connected to both $u$ and $v$ using the cumulative distribution function and $p_g$. We thereby have

$$\begin{aligned}\Pr[w\sim u,v] &= \int_0^R \rho(r)\cdot p_g(r)\,\mathrm{d}r\\ &= \Pr[r_w\leqslant r_1] + \Theta(1)\cdot\int_{r_1}^{r_0} e^{\alpha r-\alpha R}\cdot(e^{\frac{1}{2}(r_1-r)} - e^{\frac{1}{2}(r_1-r_0)})\,\mathrm{d}r\\ &= e^{\alpha r_1-\alpha R} + \Theta(1)\cdot\left[e^{\alpha r-\alpha R}\cdot(\tfrac{1}{\alpha-\frac{1}{2}}e^{\frac{1}{2}(r_1-r)} - \tfrac{1}{\alpha}e^{\frac{1}{2}(r_1-r_0)})\right]_{r_1}^{r_0}\\ &= \Theta(1)\cdot e^{\alpha r_0-\alpha R+\frac{1}{2}(r_1-r_0)}.\end{aligned}$$

Hence, the expected number of common neighbors of $u$ and $v$ is

$$c_{uv} = \Theta(1)\cdot\exp(\tfrac{R}{2} + (\tfrac{1}{2}-\alpha)r_u - \tfrac{1}{2}r_v)\cdot\Delta\theta(u,v)^{1-2\alpha}.$$

To find the angle $\varphi(c_{uv}, r_u, r_v)$ maximizing the Log-likelihood in the step model, we observe that the number of common neighbors of $u, v$ is a binomial random variable: There exists a

set $S \subseteq D_n$ in which each node is connected to both $u, v$ and each node in $D_n \setminus S$ connected to at most one of $u, v$. Since the maximum likelihood estimator for binomial random variables is the number of successes divided by the number of trials, we obtain the maximum likelihood for $\Delta\theta(u, v)$ by rearranging above equation.

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

To obtain actual values for $\Delta\theta(u, v)$ we first simply omit the constant factor hidden by $\Theta(1)$ in the above expression. To obtain reasonable angles, observe that the largest angle should likely be $\pi$. To obtain this, one can simply rescale all values of $\varphi(c_{uv}, r_u, r_v)$ with the same constant factor such that the maximum is $\pi$. As this is prone to errors if outliers exist, we instead scale all angles by the same constant such that their median is $\pi/2$. Angles that are larger than $\pi$ after this scaling are then set to $\pi$. Preliminary experiments showed that using the logarithm of the above expression for initially computing $\Delta\theta(u, v)$ (before the scaling) improved the robustness of our algorithm.

### 5.2.2 Embedding According to the Estimated Angles

In this section, we assume that we know the desired angle $\Delta\theta(u, v)$ between any pair of vertices $u$ and $v$ in the core. Our goal is to assign an angle to each vertex that realizes these differences as good as possible. To this end, we use a 1-dimensional spring embedder (see Section 3 for a short introduction to spring embedders) that basically works as follows. We start with random initial angles. Then in each iteration, we consider every pair $u, v$ of vertices. If the the current angle between $u$ and $v$ is larger than $\Delta\theta(u, v)$ we get an attractive force, otherwise we get a repulsive force. W. l. o. g., we assume $0 \leqslant \varphi_u < \varphi_v \leqslant \pi$ (the other cases work symmetrically). Moreover, let $\operatorname{err}(u, v) = \varphi_v - \varphi_u - \varphi(c_{uv}, r_u, r_v)$. The force $F_u(v)$ acting on $u$ due to $v$ is then given by

$$F_u(v) = \begin{cases} -\operatorname{err}(u,v)^2 & \text{if } \operatorname{err}(u,v) \leqslant 0, \\ \operatorname{err}(u,v)^2 & \text{if } 0 < \operatorname{err}(u,v) \leqslant \frac{\pi}{2}, \textit{and} \\ (\pi - \operatorname{err}(u,v))^2 & \text{if } \frac{\pi}{2} < \operatorname{err}(u,v) \leqslant \pi. \end{cases}$$

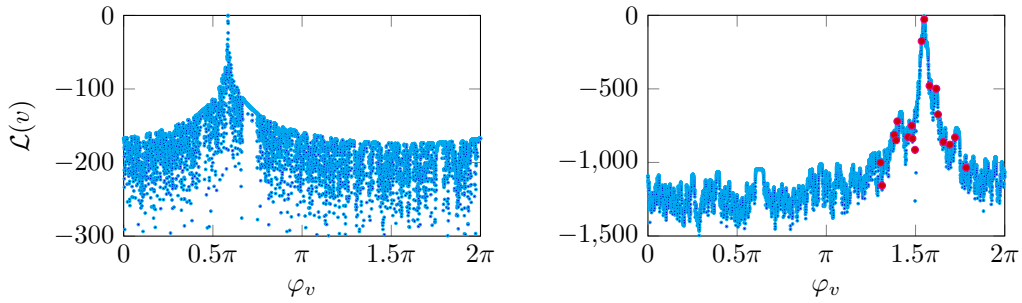
To interpret this formula, first note that $\operatorname{err}(u, v) < 0$ holds if the current angle is too small. Thus, $F_u(v)$ is negative (pushing $u$ away from $v$) and the strength of the force increases quadratically in the distance to the desired angle. Conversely, if the current angle is too large, we get a repulsive force increasing quadratically in the distance to the desired angle as long as this distance is at most $\pi/2$. For larger distances, the strength of the force actually decreases again. This has the following reason. Imagine the extreme case that $u$ and $v$ have angle $\pi$ between them but actually want to have a very small angle. Then it does not really matter whether the angle of $u$ increases or decreases as it comes closer to $v$ not matter what. Thus, we do not really want a very strong force in one of the two directions, which is the reason why we decrease the strength of attractive forces when $\operatorname{err}(u, v)$ becomes very large.

Similar to Section 3, the total force acting on $u$ is defined as

$$F_u = \sum_{v \in V \setminus u} F_u(v)$$

and the new angle of $u$ is obtained by setting $\varphi_u = \varphi_u + cF_u$. The value for $c$ is again chosen such that the maximum step size does not exceed a parameter $\theta_{\max} := \max_{u \in V}\{cF_u\}$.

Due to the 1-dimensionality of this spring embedder, we encounter a similar problem as for the hyperbolic spring embedder in Section 3: to move a vertex $u$ to a specific position,




**(a)** Exemplary fitness landscape for a node $v$ with 3 neighbors. Both methods for computing the fitness landscape exhibit no visible difference in the plot.

**(b)** Fitness landscape of a node $v$ and the coordinates at which the efficient algorithm samples the fitness. Red points indicate the sampled angles.

**Figure 2** Fitness landscape of a node $v$ computed with the efficient algorithm.

it necessarily has to pass through all vertices in between and there is no second dimension that could be used to get around them. This leads to strong repulsive forces hindering $u$ in getting to the desired position and we observed in our experiments that the algorithm often gets stuck in a local minimum. As before, we use velocity and a rather large step size $\theta_{\max}$ to circumvent this issue. Preliminary experiments showed that we obtain good results using the following parameters. We set $\theta_{\max} = 0.55\pi$ in the first iteration, decreasing it linearly down to 0 in the final iteration. For the velocity assume $F_u$ is the force from iteration $i$. Then we add $cF_u$ to the force in iteration $i+1$ where $c$ is 1 in the first iteration and linearly decreases down to 0.5 in the last iteration. Since there are $\Theta(n^{1-\alpha})$ nodes in the core [14], the total runtime of the spring embedder is $\mathcal{O}(k \cdot n^{2-2\alpha})$, where $k$ is the number of iterations. Choosing $k = \mathcal{O}(n^{2\alpha-1})$, we achieve a runtime of $\mathcal{O}(n)$.

The performance of this algorithm depends on the randomly chosen initial angles. To be able to compare core embeddings, we define a score $S$ as

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

A smaller score then indicates a better embedding. We define $s_{\text{opt}}$ as the score that is obtained when the spring embedder is initialized with the original coordinates. We then say that a core embedding is *good*, if it has a score $s \leqslant 1.2 \cdot s_{\text{opt}}$. Each graph thus has a certain probability that the core embedding is good, depending on the randomly chosen initial positions. To further increase the probability of getting a good embedding for the core, we run the spring embedder 5 times with different initial angles and use the best result, which boosts the probability of getting a good embedding to 95% for the *worst* of over 3 000 randomly generated hyperbolic random graphs (see Section 6 for the experimental setup). This suggests that the spring embedder is rather robust, i.e. we rarely encounter initial drawings that lead to bad results.

## 5.3 Computing the Log-likelihood efficiently

A key ingredient to achieve a quasilinear runtime is to improve the runtime of the Log-likelihood computation $\mathcal{L}(v)$. By a naive implementation of the Log-likelihood $\mathcal{L}(v)$ (see equation (2.2)), one needs $\Omega(n)$ time to compute the Log-likelihood of a single node. A more careful inspection, however, allows for a significant speedup.

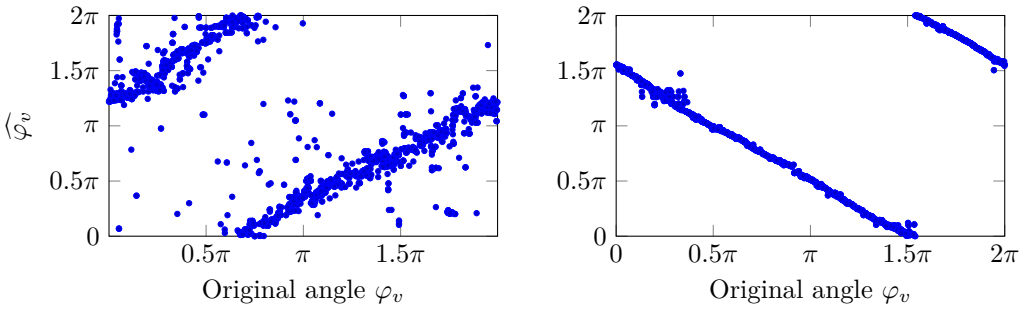



**Figure 3** The plots correspond to embeddings with average squared deviation $\Delta\varphi_G = 0.44$ (left) and $\Delta\varphi_G = 0.01$ (right). For each vertex $v$ the plot contains one point with $x$-coordinate $\varphi_v$ (angle of $v$ in the original embedding) and $y$-coordinate $\widehat{\varphi_v}$ (angle in the computed embedding).

First, observe that the total number of edges in a hyperbolic random graph is of order $\mathcal{O}(n)$; so the term $\sum_{u\in\Gamma(v)} \log(p_{uv})$ can be computed in amortized constant time. To speed up the computation of the second summand, we observe that the term $\log(1-p_{uv})$ is very close to 0 whenever $\text{dist}(u,v) \gg R$, since

$$p_{uv} := (1+\exp(\tfrac{1}{2T}(\text{dist}(u,v)-R)))^{-1} \approx \exp(-\tfrac{1}{2T}(\text{dist}(u,v)-R)),$$

and by a Taylor series for $p_{uv} \to 0$ we get

$$\log(1-p_{uv}) = -p_{uv} - \mathcal{O}(p_{uv}^2) \approx -\exp(-\tfrac{1}{2T}(\text{dist}(u,v)-R)).$$

This implies that non-neighbors that are far away from $v$ barely contribute to its Log-likelihood. If, on the other hand, $\text{dist}(u,v) \ll R$, we have $p_{uv} \to 1$, and thus

$$\log(1-p_{uv}) \approx \log(1-(1-\exp(\tfrac{1}{2T}(\text{dist}(u,v)-R)))) = \tfrac{1}{2T}(\text{dist}(u,v)-R).$$

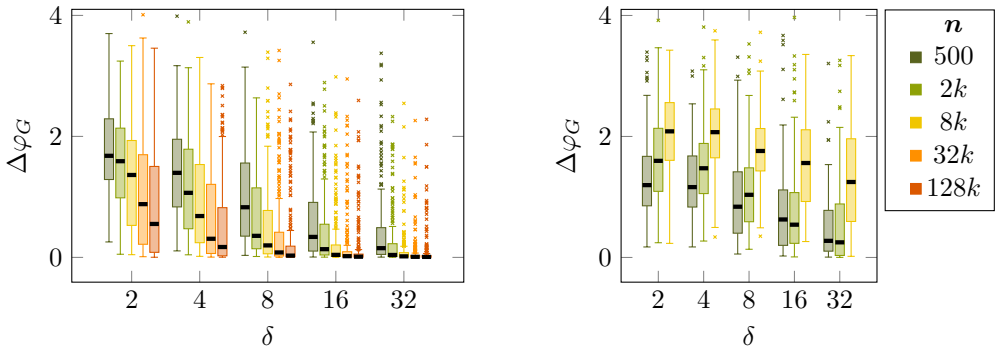
Thus, it suffices to take into account non-neighbors with low distance from $u$ while either ignoring or coarsely approximating the influence of far away non-neighbors on the Log-likelihood. To this end, we implemented the geometric data structures introduced by Bringmann et al. [9]. These were originally used to generate hyperbolic random graphs in linear time by partitioning the disc $D_n$ into suitably sized cells. To compute the Log-likelihood of a node, one can then compare it directly with nodes in neighboring cells (that have a big influence on the Log-likelihood); while averaging over all nodes in far away cells. As shown in [9], this runs in amortized time $\mathcal{O}(1)$. We need an extra $\mathcal{O}(\log n)$ factor to update the cells whenever a node is moved during the embedding algorithm.

Figure 2a shows the fitness landscapes of a node $v$; computed once via the classical exact $\Omega(n)$ method, and once using our amortized $\mathcal{O}(1)$ method. Both methods exhibit no visible differences in the plot; and we found that the relative error made by the fast Log-likelihood computation is $\leqslant 1.0025$ at all coordinates except one, where it was $\leqslant 1.02$.

## 5.4 Finding the Optimal Angle

To find a good angular coordinate for a node $v$, previous algorithms typically scan the whole range $[0, 2\pi)$ at resolution $\frac{2\pi}{n}$; and evaluate at each angle the Log-likelihood $\mathcal{L}(v)$. This incurs another factor $\Omega(n)$ on the overall runtime.

To save on this, we sample only few points around a region where a node has their maximum likelihood. To this end, we observe that the coarse likelihood landscape for a node




**(a)** Our main algorithm.

**(b)** Our hyperbolic spring embedder.

**Figure 4** Each data point in the box plot represents the value of $\Delta\varphi_G$ for a single graph $G$ ($y$-axis) depending on the average degree $a$ ($x$-axis). The graphs are grouped into small, medium, and large graphs.

$v$ (for small $T$) is governed by the position of $v$'s neighbors. Furthermore, neighbors with large radii have a larger influence on the fitness landscape, as the hyperbolic distance to these nodes increases more quickly than to neighbors with small radial coordinates. Hence, $v$ needs to be placed close to its embedded low-degree neighbors.

Ignoring non-neighbors for now, we achieve this by computing a weighted average over the angles of all neighbors of $v$. Let $u_1, \dots, u_k$ be the embedded neighbors of $v$. Then, $v$'s angle is computed as follows.

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

To take non-neighbors into consideration, we then randomly sample $\mathcal{O}(\log(n))$ points around this angle and use the one with the smallest Log-likelihood. Figure 2b shows the fitness landscape of an exemplary node $u$, as well as the randomly sampled angles. As can be seen, the heuristic typically finds good candidates whose angles are close to the optimal angle.

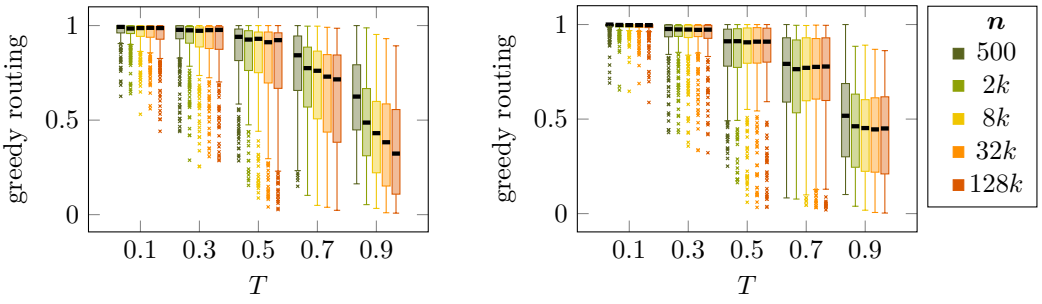
## 6 Experiments

To evaluate the quality of our algorithm, we sampled 10 different graphs for every combination of the following parameters: $\alpha \in \{0.55, 0.65, 0.75, 0.85, 0.95\}$, $T \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, $\delta \in \{2, 4, 8, 16, 32\}$, $n \in \{500, 2\,000, 8\,000, 32\,000, 128\,000\}$. This results in a total of 6 250 graphs. For each of these graphs, we computed the following statistics: Log-likelihood, success ratio of greedy routing and the average squared deviation in the original angle vs. estimated angle plot. We present the most insightful statistics in standard box plot form.[2]

### 6.1 Quality

A popular way to judge whether an embedding makes sense is to plot the embedded angular coordinates against the original generated coordinates. If the result resembles a straight line

[2] A box contains 50% of all data points; the median is marked black. Points are considered outliers if they have distance more than 1.5× IQR to the box. The whiskers depict the closest data point to the box that is not an outlier.




**(a)** Our main algorithm.

**(b)** Originally generated embeddings.

**Figure 5** The success ratio of greedy routing ($x$-axis) depending on the value of $T$ ($y$-axis) grouped with respect to the number of vertices (colors).

(that might have a cyclic shift), then the relative ordering of nodes has been reconstructed well in the embedding. Two examples for such plots are shown in Figure 3. To allow for comparisons that scale to a large amount of graphs, we derive the following quality measure. For a vertex $v$ let $\Delta\varphi_v$ be the quadratic difference between $\varphi_v$ in the original embedding and $\varphi_v$ in the computed embedding. For a graph $G = (V, E)$, the value $\Delta\varphi_G = \sum_{v \in V} \Delta\varphi_v / n$ then describes the average squared deviation in $G$.

The box plot in Figure 4a plots $\Delta\varphi_G$ against the average degree $\delta$; grouped by the size of the graph. In this and all other plots, we average over all parameters that are not explicitly grouped by. Observe that $\Delta\varphi_G$ is high if the average degree is small, as the few existing edges are not sufficient to uniquely determine the single best embedding. Thus, several embeddings may be equally good. In fact, for small $\delta$, our algorithm finds an embedding with a Log-likelihood very close to the Log-likelihood of the original embedding (the mean values for large graphs with $\delta = 2$ are $-2.39 \cdot 10^5$ for the embedding and $-2.19 \cdot 10^5$ for the original, respectively, while the corresponding values for $\delta = 16$ are $-1.78 \cdot 10^6$ and $-1.16 \cdot 10^6$). For an average degree of 8, the mean value for $\Delta\varphi_G$ of all medium sized and large graphs is 0.2 and 0.04, respectively. For comparison, note that the plots in Figure 3 correspond to graphs with values 0.01 and 0.44. Also note that our algorithm performs particularly well on large graphs, which was the goal we aimed for.

For comparison with the spring embedder described in the online version, see Figure 4b. As the spring embedder is too slow on large graphs, we only ran the experiments on medium and small graphs. Note that the quality of the spring embedder decreases for increasing graph size. In contrast, it performs comparatively well on small graphs (and in some cases actually better than our main algorithm) while it is heavily outperformed on the medium sized graphs. Hence, the spring embedder is a reasonable option for graphs with up to 1 000 vertices, while our main algorithm is the better option for larger graphs.

A quality measure previously used for hyperbolic embeddings is the success ratio of greedy routing. Figure 5a shows this ratio for the embeddings generated by our algorithm depending on the parameter $T$, grouped by the size of the graph. Observe that the ratio is close to 100% for small values of $T$ but drops significantly for larger values. This is unfortunate as real world graphs are considered to have fairly large values of $T$, e.g., $T = 0.7$ was used for the embedding of the Internet graph [8]. Though this particular embedding allows greedy routing with success ratio 97%, the ratios of around 80% we obtain for $T = 0.7$ seem to reflect the typical behavior of random hyperbolic graphs much better; see Figure 5b.

Note that these observations imply that maximizing the Log-likelihood will not necessarily lead to the desired result in terms of greedy routing. Conversely, optimizing the embedding

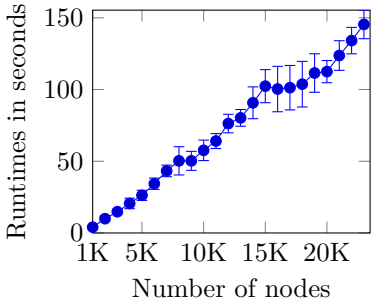



**Figure 6** Runtimes for the embedding algorithm. Error bars show the standard deviation.

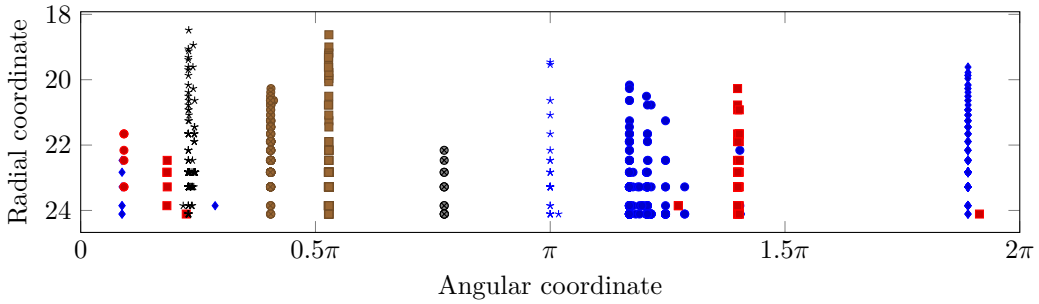



**Figure 7** The nine largest communities in the amazon product recommendation network. For clarity, only nodes that belong to a single community are shown. Nodes belonging to the same community are typically placed nearby, even though the embedding algorithm had no knowledge of the ground truth communities.

for greedy routing will probably not lead to an embedding that is close to the original embedding of a hyperbolic random graph. Hence, we do not see the low success ratios our embeddings achieve for large $T$ as a weakness but rather as a strength as it matches the behavior of the original embedding.

## 6.2 Runtime

A key contribution of our algorithm is its significant improvement on the runtimes compared to previous approaches. The runtime experiments were performed on commodity hardware, i.e. a 2.7 GHz Core i7 with 8 GB of RAM. Figure 6 shows the runtimes depending on $n$. Note that compared to available algorithms these are fairly quick: Graphs of size 20 000 can be embedded in under two minutes. We even embedded graphs of size 330 000 in under one hour, see Section 6.3. For comparison, the reference algorithm HyperMap [27, 29] needs over 1.5 hours for a graph of size 2 000.

## 6.3 Embedding a Real-World Graph

As a proof of concept, we embed the Amazon product recommendation network [41]. It has $n = 334\,863$ nodes with an average degree of 5.53, the degree distribution follows a power-law with exponent $\beta = 3.6$ and the average clustering coefficient is 0.4. The nodes represent products available on Amazon, and an edge $\{u, v\}$ is present if product $u$ is recommended together with product $v$. Product categories define ground truth communities in this graph.

The embedding took 50 minutes on a single 2.7 GHz Core i7. While the number of nodes is too large to visually inspect the whole graph, we have plotted the nine largest communities in Figure 7. Most nodes belonging to a single community are mapped close together; which suggests that the hyperbolic embedding might be a useful tool in discovering hidden communities in a large network.

**Acknowledgements.** We thank the authors of [27] for their code and for helpful discussions; Christoph Kessler (HPI Potsdam) and Maximilian Katzmann (FSU Jena) for their help with the experiments; and Konrad Schöbel (FSU Jena) for fruitful discussions on hyperbolic variants of MDS.

## References

1 William Aiello, Fan Chung, and Linyuan Lu. A random graph model for massive graphs. In *32nd Symp. Theory of Computing (STOC)*, pages 171–180, 2000.

2 William Aiello, Fan Chung, and Linyuan Lu. A random graph model for power law graphs. *Experimental Mathematics*, 10(1):53–66, 2001.

3 Rodrigo Aldecoa, Chiara Orsini, and Dmitri Krioukov. Hyperbolic graph generator. *Computer Physics Communications*, 196:492–496, 2015. doi:10.1016/j.cpc.2015.05.028.

4 Dena Marie Asta and Cosma Rohilla Shalizi. Geometric network comparisons. In *31st Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 102–110, 2015.

5 Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

6 Michel Bode, Nikolaos Fountoulakis, and Tobias Müller. On the giant component of random hyperbolic graphs. In *7th European Conf. Combinatorics, Graph Theory and Applications*, pages 425–429, 2013.

7 Michel Bode, Nikolaos Fountoulakis, and Tobias Müller. The probability that the hyperbolic random graph is connected. www.math.uu.nl/~Muell001/Papers/BFM.pdf, 2014.

8 Marián Boguñá, Fragkiskos Papadopoulos, and Dmitri Krioukov. Sustaining the internet with hyperbolic mapping. *Nature Communications*, 1:62, 2010.

9 Karl Bringmann, Ralph Keusch, and Johannes Lengler. Geometric inhomogeneous random graphs. *arXiv preprint arXiv:1511.00576*, 2015.

10 F. Chung and L. Lu. Connected components in random graphs with given expected degree sequences. *Annals of Combinatorics*, 6(2):125–145, 2002.

11 Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

12 James R. Clough and Tim S. Evans. Embedding graphs in lorentzian spacetime. *arXiv 1602.03103*, 2016.

13 Trevor F. Cox and M.A.A. Cox. *Multidimensional Scaling, Second Edition.* Chapman and Hall/CRC, 2 edition, 2000.

14 Tobias Friedrich and Anton Krohmer. Cliques in hyperbolic random graphs. In *34th IEEE Conf. Computer Communications (INFOCOM)*, pages 1544–1552, 2015.

15 Tobias Friedrich and Anton Krohmer. On the diameter of hyperbolic random graphs. In *42nd Intl. Coll. Automata, Languages and Programming (ICALP)*, pages 614–625, 2015.

16 Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. Random hyperbolic graphs: degree sequence and clustering. In *39th Intl. Coll. Automata, Languages and Programming (ICALP)*, pages 573–585, 2012.

17 Stephen G. Kobourov. *Handbook of Graph Drawing and Visualization*, chapter Force-Directed Drawing Algorithms, pages 383–408. Chapman and Hall/CRC, 2013.

**18** Stephen G. Kobourov and Kevin Wampler. Non-eeuclidean spring embedders. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):757–767, 2005.

**19** Christoph Koch and Johannes Lengler. Bootstrap percolation on geometric inhomogeneous random graphs. In *43rd Intl. Coll. Automata, Languages and Programming (ICALP)*, 2016.

**20** Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Phys. Rev. E*, 82:036106, 2010.

**21** John Lamping, Ramana Rao, and Peter Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *13th ACM CHI*, pages 401–408, 1995. doi:10.1145/223904.223956.

**22** Jonh Lamping and Ramana Rao. The hyperbolic browser: A focus+context technique for visualizing large hierarchies. *Journal of Visual Languages & Computing*, 7(1):33–55, 1996.

**23** David Liben-Nowell and Jon M. Kleinberg. The link-prediction problem for social networks. *J. Am. Soc. Inf. Sci.*, 58(7):1019–1031, 2007. doi:10.1002/asi.20591.

**24** Tamara Munzner. Exploring large graphs in 3d hyperbolic space. *IEEE Computer Graphics and Applications*, 18(4):18–23, 1998. doi:10.1109/38.689657.

**25** M. E. J. Newman. Clustering and preferential attachment in growing networks. *Phys. Rev. E*, 64:025102, 2001.

**26** Ilkka Norros and Hannu Reittu. On a conditionally Poissonian graph process. *Advances in Applied Probability*, 38(1):59–75, 2006.

**27** Fragkiskos Papadopoulos, Rodrigo Aldecoa, and Dmitri Krioukov. Network geometry inference using common neighbors. *Phys. Rev. E*, 92:022807, Aug 2015. doi:10.1103/PhysRevE.92.022807.

**28** Fragkiskos Papadopoulos, Maksim Kitsak, M Ángeles Serrano, Marián Boguñá, and Dmitri Krioukov. Popularity versus similarity in growing networks. *Nature*, 489(7417):537–540, 2012.

**29** Fragkiskos Papadopoulos, Constantinos Psomas, and Dmitri V. Krioukov. Network mapping by replaying hyperbolic growth. *IEEE/ACM Trans. Netw.*, 23(1):198–211, 2015. doi:10.1109/TNET.2013.2294052.

**30** Mathew D. Penrose. *Random Geometric Graphs*. Oxford University Press, 2003.

**31** Yuval Shavitt and Tomer Tankel. Hyperbolic embedding of internet graph for distance estimation and overlay construction. *IEEE/ACM Trans. Netw.*, 16(1):25–36, 2008. doi:10.1145/1373452.1373455.

**32** Eleni Stai, Vasileios Karyotis, and Symeon Papavassiliou. A hyperbolic space analytics framework for big network data and their applications. *IEEE Network*, 30(1):11–17, 2016. doi:10.1109/MNET.2016.7389825.

**33** Remco van der Hofstad. Random graphs and complex networks. Available at www.win.tue.nl/~rhofstad/NotesRGCN.pdf, 2011.

**34** Alexei Vázquez. Growing network with local rules: Preferential attachment, clustering hierarchy, and degree correlations. *Phys. Rev. E*, 67:056104, May 2003. doi:10.1103/PhysRevE.67.056104.

**35** Kevin Verbeek and Subhash Suri. Metric embedding, hyperbolic space, and social networks. In *30th Annual Symposium on Computational Geometry (SOCG)*, page 501, 2014. doi:10.1145/2582112.2582139.

**36** Moritz von Looz, Henning Meyerhenke, and Roman Prutkin. Generating random hyperbolic graphs in subquadratic time. In *26th Intl. Symp. Algorithms and Computation (ISAAC)*, pages 467–478. Springer, 2015.

**37** Jörg A. Walter. H-MDS: a new approach for interactive visualization with multidimensional scaling in the hyperbolic space. *Inf. Syst.*, 29(4):273–292, 2004. doi:10.1016/j.is.2003.10.002.

**38** Jörg A. Walter and Helge J. Ritter. On interactive visualization of high-dimensional data using the hyperbolic plane. In *8th ACM Intl. Conf. Knowledge Discovery and Data Mining (SIGKDD)*, pages 123–132, 2002. doi:10.1145/775047.775065.

**39** Zuxi Wang, Qingguang Li, Fengdong Jin, Wei Xiong, and Yao Wu. Hyperbolic mapping of complex networks based on community information. *Physica A: Statistical Mechanics and its Applications*, 455:104–119, 2016.

**40** Kilian Q Weinberger and Lawrence K Saul. Unsupervised learning of image manifolds by semidefinite programming. *International Journal of Computer Vision*, 70(1):77–90, 2006.

**41** Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

**42** Xiaohan Zhao, Alessandra Sala, Haitao Zheng, and Ben Y. Zhao. Efficient shortest paths on massive social graphs. In *7th International Conference on Collaborative Computing (CollaborateCom)*, pages 77–86, 2011.