Efficient Embedding of Scale-Free Graphs in the Hyperbolic Plane

Thomas Bläsius, Tobias Friedrich, Anton Krohmer, and Sören Laue

***Abstract*— Hyperbolic geometry 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 at least a quadratic runtime. Our algorithm achieves quasi-linear 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, such as log-likelihood and greedy routing, our algorithm discovers embeddings that are very close to the ground truth.**

***Index Terms*— Applications, inference, network geometry.**

I. 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, such as a power law degree distribution, large 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 [2]. The inhomogeneous random graph model by van der Hofstad [3] provides the greatest accessibility for rigorous mathematical analysis. It also generalizes the models of Chung and Lu [4] and Aiello *et al.* [5], [6] and Norros and Reittu [7].

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 for real-world networks. For social networks the reason is easy to see. It is more likely for two persons to be friends if they already have friends in common than it would be for two random strangers to forge a connection. There are a number of modifications to the above models that incorporate this intuition [8]–[10], however, all of these fixes introduce other artificial artifacts and cannot explain *why* the clustering occurs in the first place.

Manuscript received January 27, 2017; revised September 17, 2017 and January 29, 2018; accepted February 9, 2018; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Andrews. Date of publication March 12, 2018; date of current version April 16, 2018. The work of T. Friedrich and S. Laue was supported by the German Science Foundation under Grant FR-2988 and Grant LA-2971. A preliminary version of this paper appeared in [1]. *(Corresponding author: Anton Krohmer.)*

T. Bläsius, T. Friedrich, and A. Krohmer are with the Department of Algorithm Engineering, Hasso Plattner Institute, 14482 Potsdam, Germany (e-mail: anton.krohmer@hpi.de).

S. Laue is with the Chair of Theoretical Computer Science II, University of Jena, 07743 Jena, Germany (e-mail: soeren.laue@uni-jena.de).

Digital Object Identifier 10.1109/TNET.2018.2810186

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 known that geometric random graphs with an underlying Euclidean space result in a Poisson degree distribution [11]. Krioukov *et al.* [12] 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 at a small (hyperbolic) distance. 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 [12], [13], to have a polylogarithmic diameter and ultra-short average distances of order $\mathcal{O}(\log \log n)$ [14], [15], and to allow fast bootstrap percolation [16].

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 algorithmic problem addressed, efficiently *generates* such a graph or, equivalently, *samples* a graph from the probability distribution defined by hyperbolic random graphs. The naive generation of a hyperbolic random graph takes $\Theta(n^2)$ time [17]. Using a polar quadtree adapted to hyperbolic space, von Looz *et al.* [18] achieved a time complexity of $\mathcal{O}((n^{3/2} + m) \log n)$. By a more sophisticated partitioning of the space, Bringmann *et al.* [15] obtained an optimal expected linear runtime for generation, which is crucial for large-scale experiments.

Embedding Networks Into Hyperbolic Geometry

It is well known in the visualization community that hierarchical or tree-like structures can be well represented in a hyperbolic space [19]. This mainly comes from the fact that the volume of hyperbolic space expands exponentially, compared to polynomial expansion in Euclidean space. Another application for hyperbolic embeddings arises from the fact that the hyperbolic geometry appears to be well suited for greedy routing [20]. Moreover, our experiments in Section VI-D suggest that hyperbolic embeddings can be useful for community

1063-6692 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

detection or link prediction. There are three general 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 [21]–[23]. As trees can be embedded perfectly, this is a very efficient way to map a network and has been used for interactive network browsers. It allows assigning more display space to the interesting portions of a network [24], [25]. 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 graph distances. Embedding the all-pair-shortest-path matrix can be done with the well established Euclidean data analysis method Multidimensional Scaling (MDS) [26], which has been translated to hyperbolic geometry [27]. Due to the quadratic size of the distance matrix, this approach only works in practice for graphs with a few hundred nodes [28]. To reduce the runtime, it is possible to (randomly) select a small subset of the pairwise distances [29]–[31].
- 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 [32]. The reason why connections between vertices exist can be twofold. On the one hand, the two vertices may be similar, which holds e. g. for close friends in social networks or for geographically close autonomous systems (AS) 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 these details 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.* [12].

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.* [20] 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. On average, such greedy paths are just 10% longer than the shortest paths found in the network. However, the described method to discover the hyperbolic coordinates "require[s] significant manual intervention [...] to lead to any reasonable results in a reasonable amount of compute time" [33]. A general algorithm for embedding a network in a hyperbolic space was later presented by Papadopoulos *et al.* [33]. 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 $\mathcal{O}(n^2)$ [34]. Papadopoulos *et al.* [33], [34] present no runtime measurements, but their HyperMap code on our machine requires more than 1.5 hours for a graph of size $2\,000$ (cf. Section VI-C).

Improvements to HyperMap have been suggested. For instance, Wang *et al.* [35] use a community detection algorithm for the coarse layout of the nodes and an MLE to find precise positions. Alanis-Lobato *et al.* [36] take a different approach by embedding the graph using its Laplacian and combine it with HyperMap for improved performance [37]. Both these methods, however, still require a running time of $\Omega(n^2)$.

Our New Hyperbolic Embedder

We design and implement a new algorithm for computing hyperbolic MLE embeddings of massive networks (Section V). Our code is available online.[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 [34], 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.* [15] that allows traversing nodes of close proximity in expected amortized constant time.

This enables us to embed significantly larger graphs than before. For instance, in under one hour we computed on commodity hardware a hyperbolic embedding of the Amazon product recommendation network that 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 ground truth data (Section VI). 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 III and IV). We find that both of them can work under some strong assumptions but generally fail to translate to large real-world graphs. Though there are some fundamental difficulties with spring embedders in the hyperbolic plane, a spring embedder will prove useful as a subroutine in our main algorithm in Section VI.

[1] https://hpi.de/friedrich/research/hyperbolic

II. Preliminaries

In this section, we briefly introduce the hyperbolic random graph model. We keep the definitions concise and refer the reader to previous work for a more intuitive introduction, see e.g. [12], [13]. We use the native representation of the hyperbolic space [12] of curvature -1, where points are identified by radial coordinates (r, φ). The first coordinate describes the hyperbolic distance from the origin, and two points x, y have hyperbolic distance

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

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 disk D_R 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, φ_v) sampled from the probability density function $f(r,\varphi) = \frac{\alpha \sinh(\alpha r)}{2\pi(\cosh(\alpha R)-1)}$, where α is a parameter adjusting the power law exponent $\beta = 2\alpha + 1$ of the resulting network. Then, every two vertices u, v are connected with a probability p depending on their distance:

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

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), α, and T. Following standard graph notation, we write $\Gamma(v)$ for the set of neighbors of v, and we use δ 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 judge the quality of this embedding using the *Log-likelihood*

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

where the hyperbolic distances dist are taken with respect to the coordinates $\{r_i, \varphi_i\}_{i=1}^n$. Observe that $\exp(\mathcal{L}(\{r_i, \varphi_i\}_{i=1}^n \mid G))$ is exactly the probability to generate graph G with the hyperbolic random graph model, conditioned on node i having position $\{r_i, \varphi_i\}$ for all i.

To determine the quality of a specific node v, we write

$$\mathcal{L}(v) := \sum_{u\in\Gamma(v)} \log(p(\operatorname{dist}(u,v))) + \sum_{u\notin\Gamma(v)} \log(1 - p(\operatorname{dist}(u,v))), \quad (3)$$

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 output hyperbolic coordinates close to the original embedding. As an additional requirement, we would like that the algorithm is robust to noise, that is, it 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.

III. Spring Embedder

A heavily used technique to embed graphs in the Euclidean plane is the force-directed method (also called spring embedder) [38], 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 apart. 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 [39]. In the following, we discuss why a straightforward implementation of a spring embedder in the hyperbolic plane does not work in our setting. In Section III-B we present several adaptations that lead to good results, at least for smaller graphs.

A. 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 stable except that v is far away from u. Now when v moves towards u, it also approaches other vertices it is not connected to, which then push v back towards the direction it came from. This is not a problem, however, as there are usually only a few vertices close enough to v for their force to be 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 radial coordinate, 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 a larger radius. Thus, even bad embeddings are stable.

Judging from the pictures presented by Kobourov and Wampler [39], 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

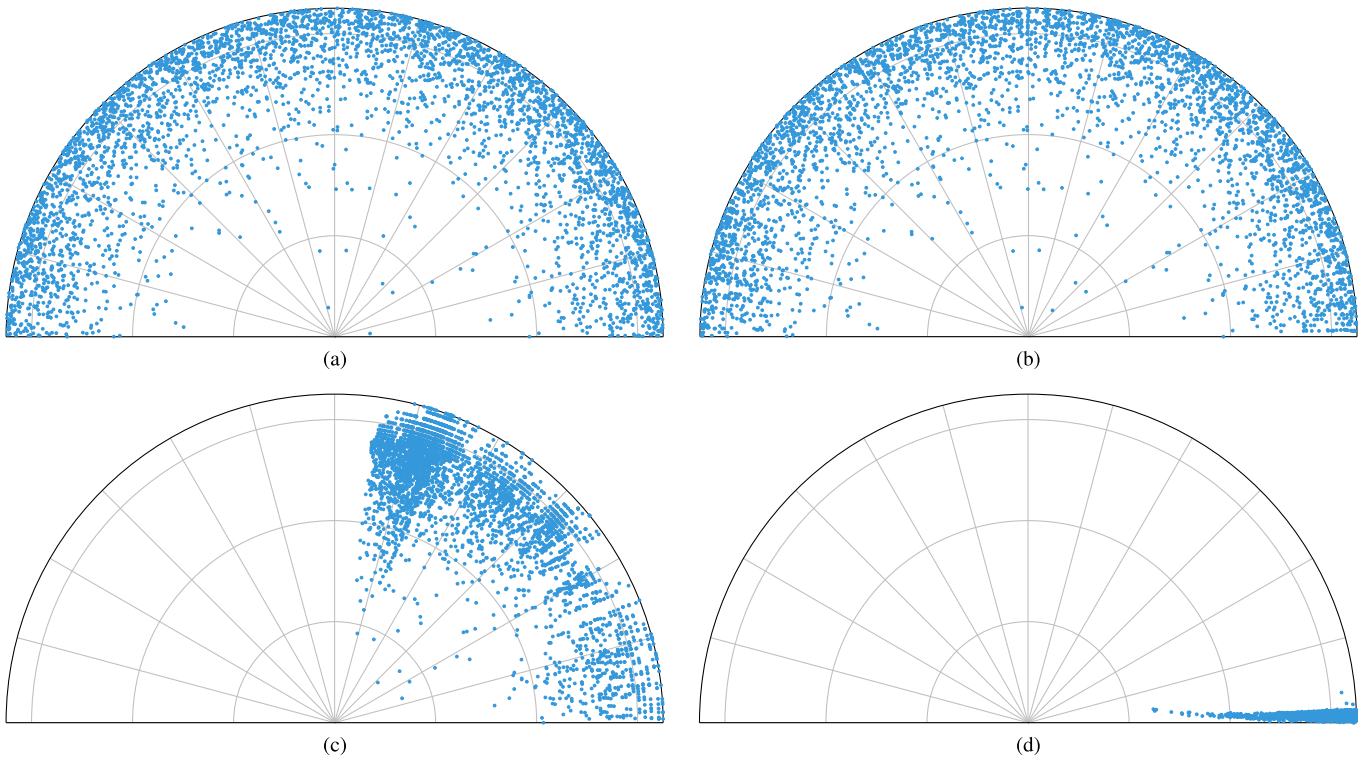

Fig. 1. First phase of the LP from Section IV. Since nodes are placed in $[0, \pi]$, half of D_R is hidden. (a) Original Points (edges not shown) of a hyperbolic random graph with $T = 0$. (b) Embedded nodes using the LP. All parameters except the angular coordinates were given as additional information. The embedding is almost equivalent to the original. (c) Embedded nodes using the LP with estimated radial coordinates (See Section V-A). 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.

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.

B. Fixing the Spring Embedder

We circumvent the above described problems by treating the two components of a coordinate (i.e., the radius and angle) more or less independently. More precisely, let u and v be two vertices. Assume without loss of generality that $0 \leqslant \varphi_u < \varphi_v \leqslant \pi$, that is, increasing φ_u moves u towards v. We define the forces $F^u_\varphi(v)$ and $F^u_r(v)$ acting on the angle and on the radius, respectively, as

$$F^u_\varphi(v) = \begin{cases} 1 - p(\mathrm{dist}(u,v)) & \text{if } \{u,v\} \in E, \\ -p(\mathrm{dist}(u,v)) & \text{otherwise,} \end{cases}$$

and

$$F^u_r(v) = \begin{cases} -(1 - p(\mathrm{dist}(u,v))) & \text{if } \{u,v\} \in E, \\ p(\mathrm{dist}(u,v)) & \text{otherwise.} \end{cases}$$

Recall that $p(\mathrm{dist}(u,v))$ denotes the probability that u and v with hyperbolic distance $\mathrm{dist}(u,v)$ are adjacent. The total forces F^u_φ and F^u_r for the vertex u are defined as

$$F^u_\varphi = \sum_{v \in V \setminus \{u\}} F^u_\varphi(v), \quad \text{and } F^u_r = \sum_{v \in V \setminus \{u\}} F^u_r(v).$$

After these forces are computed for each vertex $u \in V$, it is moved from (r_u, φ_u) to $(r_u + c_r F^u_r, \varphi_u + c_\varphi F^u_\varphi)$. The values for c_φ and c_r are chosen such that $\max_{u \in V}\{c_\varphi F^u_\varphi\} = \varphi_{\max}$ and $\max_{u \in V}\{c_r F^u_r\} = r_{\max}$ holds for the parameters $\varphi_{\max}$ and $r_{\max}$, which basically ensures that no angle and no radius is changed by more than $\varphi_{\max}$ and $r_{\max}$, respectively.

Note that $F^u_\varphi(v)$ is positive if u and v are adjacent and thus $F^u_\varphi(v)$ contributes to decreasing the angle between u and v (as we assumed $0 \leqslant \varphi_u < \varphi_v \leqslant \pi$), which coincides with the desired behaviour. On the other hand $F^u_r(v)$ is always negative if u and v are connected and positive otherwise. This can have the counter-intuitive effect that v contributes to moving u towards the origin although u and v are connected and v is farther away from the origin than u, which increases the difference between their radii. However, unless u and v have almost the same angle, this actually moves u closer to v (with respect to hyperbolic distance) and thus has the desired effect.

Before we discuss the choices for the parameters $\varphi_{\max}$ and $r_{\max}$, we want to point out some potential issues (and how to fix them). First note that edge probability $p(d(u,v))$ depends on the radius R and on the parameter T, both of which we estimate as described in Section V-A. Note that for $T \to 0$ (or for constant T with increasing R), the edge probability converges to the step function, that is, $p(d(u,v)) \to 1$ if $d(u,v) \leqslant R$ and $p(d(u,v)) \to 0$ otherwise. This has two undesirable effects. First, if u and v are only just close enough (in case they are adjacent) or only just sufficiently far apart (in case they are not connected), then there are no forces that

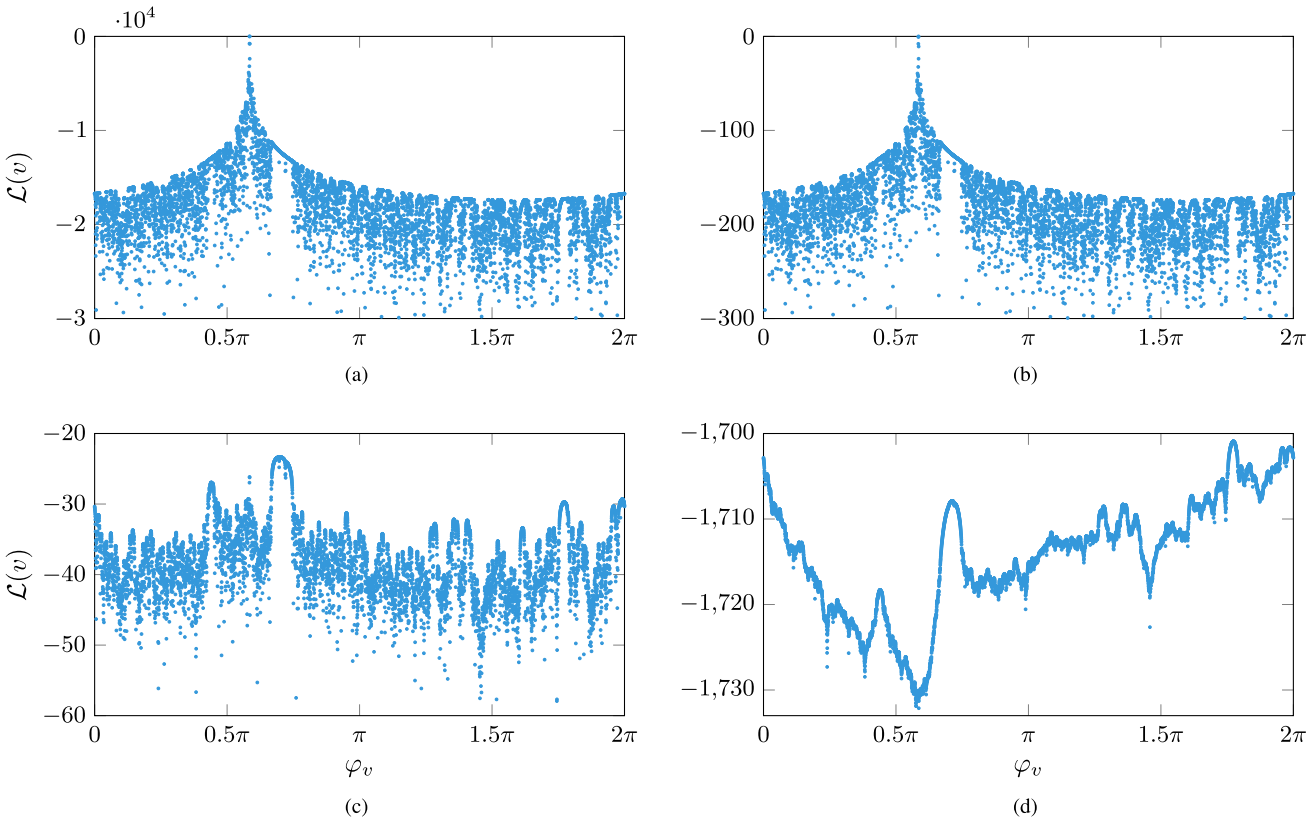

Fig. 2. Fitness landscape plot of a node v for different values of T. The x-axis shows candidate angular positions for node v, the y-Axis shows the Log-likelihood $\mathcal{L}(v)$ at that angle. (a) $T = 0.001$. (b) $T = 0.1$. (c) $T = 1$. (d) $T = 10$.

work towards keeping this situation like this. Second, vertices that are way too close (but not adjacent) or way too far apart (but adjacent) have roughly the same influence as vertices that are only slightly too close or slightly too far apart. Both effects are especially problematic in the early stages of the algorithm. To resolve this issue, we start with the rather large value $T = 0.3\ R$ in the first iteration and decrease it linearly. More precisely, in the ith iteration out of I iterations in total, we set $T = 0.3\ R \times \max\{0.05, (1 - i/I)\}$. We note that using a linear dependency on R is reasonable as this leads to roughly the same shape of the function $p(xR)$ for $x \in [0, 2]$ independent of R.

As a second potential issue, note that handling the angle independently from the radius leads to huge jumps in terms of hyperbolic distance for vertices with large radius (unless $\varphi_{\max}$ is unreasonably small). Such large jumps are usually undesirable in the Euclidean plane (and do not occur without a large change to at least one coordinate). In the hyperbolic plane, we however allow these large jumps since prohibiting them them leads to exactly the problems described in Section III-A.

For the final issue, which also leads us to the parameters $\varphi_{\max}$ and $r_{\max}$, first consider a vertex u moving through the Euclidean plane towards its desired location. If there is another non-adjacent vertex v on its way, then getting close to v leads to potentially large repulsive forces. However, u usually does not get stuck because of these forces as u and v can get around each other by a slight movement in opposite directions orthogonally to the actual movement of u. In the hyperbolic plane, two vertices with the same angle are close to each other no matter what their radius is. Thus, while changing the angle of u to get it to its desired value, u necessarily comes close to every other vertex whose angle is between φ_u and the position u aims for. Thus, the algorithm is much more likely to get stuck in a local minimum than a spring embedder in the Euclidean plane.

We use two strategies to circumvent this issue. The first is to simply allow rather large changes to the coordinates (i.e., use large values for $\varphi_{\max}$ and $r_{\max}$), which makes it possible to jump out of local minima. To make sure that the algorithm still converges to a stable position, we decrease $\varphi_{\max}$ and $r_{\max}$ for later iterations. More precisely, we use $\varphi_{\max} = \pi$ and $r_{\max} = R$ in the first iteration and decrease both values linearly down to 0.

The second strategy is to simulate some kind of velocity. In the above example, this can help u to get past v as the repulsive force of v may slow u down instead of actually pushing it back to where it came from. A simple way to achieve such a notion of velocity is as follows. Assume F^u_φ is the force acting on u in iteration i. Then in iteration $i + 1$, we compute the new force as before and add cF^u_φ to it, where c is 1 in the first iteration and decreases linearly down to 0.2 in the last iteration.

To conclude this section, we have seen that there are several reasons why spring embedders work less well in the hyperbolic plane than in the Euclidean plane. We suggested potential solutions for these problems and we see in Section VI that our spring embedder actually performs reasonably well at least on small to medium sized instances. Moreover, we see in Section V-B how techniques described above can be reused to embed the core of a larger graph.

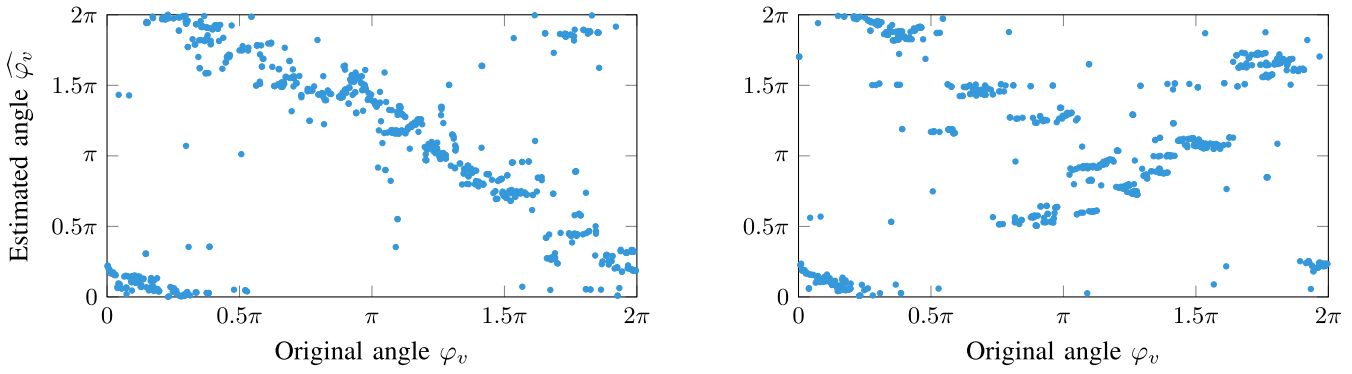

Fig. 3. Original angular coordinates vs. embedded angular coordinates for a generated hyperbolic random graph with $T = 0.7$. (a) Our algorithm sets $T = 0.1$ for any input and is able to reconstruct the original ordering of nodes fairly well. (b) Choosing $T = 0.7$ and scanning the whole range $[0, 2\pi)$ for the best Log-likelihood of a node results in a much worse embedding.

IV. 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.

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:

$$\begin{aligned}
&\text{max.} \sum_{j=1}^{n} \varphi_j \\
&\text{s.t. } \varphi_i - \varphi_j \leqslant \theta(r_i, r_j), \quad i,j = 1,\dots,n, \quad \text{if } \{i,j\} \in E \\
&\qquad \varphi_j - \varphi_i \leqslant \theta(r_i, r_j), \quad i,j = 1,\dots,n, \quad \text{if } \{i,j\} \in E \\
&\qquad 0 \leqslant \varphi_i \leqslant pi \quad i = 1,\dots,n \\
&\qquad \varphi_v = 0, \quad \text{for some starting node } v
\end{aligned}$$

where $\theta(r_i, r_j)$ is the maximal angular distance such that two nodes with radii r_i, r_j are still connected. Formally,

$$\theta(r_i, r_j) = \arccos\left(\frac{\cosh(r_i)\cosh(r_j) - \cosh(R)}{\sinh(r_i)\sinh(r_j)}\right). \quad (4)$$

The LP has a caveat: it is only able to spread nodes on the half circle $[0, \pi]$. For larger angular coordinates, the hyperbolic distances start decreasing again, which is not encodable in the LP. This problem, however, can be remedied by 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_R: If w has an angular distance of 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 Figures 1a and 1b). It is, however, extremely volatile to the quality of the estimated parameters. In addition, it fails completely when used on a real graph or even a graph generated by the binomial model (see Figures 1c and 1d). 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.

V. The Embedder

Our embedding algorithm is inspired by the Metropolis-Hastings Algorithm from [20]. Algorithm 1 contains a bird's eye view of all steps. Detailed descriptions of the individual steps follow in the next sections.

Algorithm 1 Fast Embedding Algorithm

Input: Undirected connected Graph $G = (V, E)$
Output: Hyperbolic coordinates $(r_i, \varphi_i)_{i=1}^{\hat{n}}$ $(\hat{n} = |V|)$

```
Estimate global parameters n, R, α, T
Estimate radial coordinates r_i                          ▷ See Sec. V-A
for all nodes v ∈ V do
    Place v in layer L_i if deg(v) ∈ [2^i, 2^{i+1} − 1]
Embed all nodes in layers ⩾ (log n)/2                    ▷ See Sec. V-B
for i = (log n)/2 − 1 ... 0 do                           ▷ See Sec. V-C, V-D
    for log n times do
        for all v ∈ ⋃_{j⩾i} L_j do
            Embed v by optimizing its Log-likelihood
```

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 V-A. This step can be done in linear time.

In the second phase, nodes are grouped into layers by their degree, and then embedded layer by layer. For bootstrapping, high-degree nodes (inner layers) 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 low-degree nodes in subsequent layers are typically placed close to their neighbors in previously embedded layers. This step is described in Section V-B. As the size of the subgraph embedded in this step is significantly sublinear,

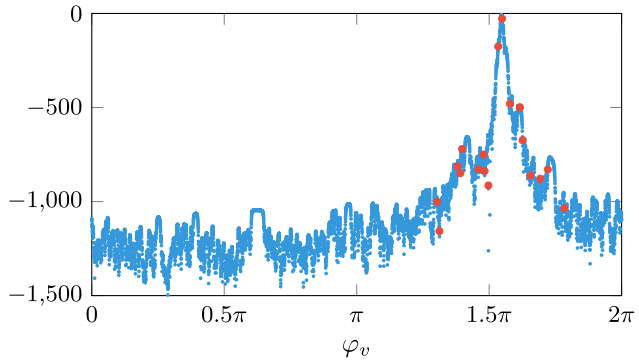

Fig. 4. Fitness landscape of a node v and the coordinates at which the efficient algorithm samples the fitness. Red points indicate the sampled angles.

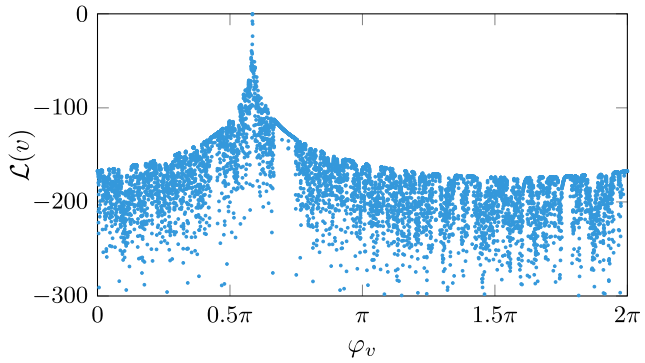

Fig. 5. Exemplary fitness landscape for a node v with 3 neighbors. Both methods for computing the fitness landscape exhibit no visible difference in the plot.

we can spend more time in this step and still obtain a linear runtime.

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 V-C and V-D. This is repeated $\log n$ times per layer (and there are only $\mathcal{O}(\log n)$ layers). While this step is similar to HyperMap [20], [33], [34], we improve upon their algorithm by achieving an amortized polylogarithmic runtime per node as compared to their linear runtime. Thus, our overall algorithm thus runs in $\mathcal{O}(n \cdot \text{polylog}(n))$.

A. 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 disk D_R, the parameter α 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 (2)) 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.

Hyperbolic random graphs, however, are typically disconnected. For power law exponents $2 < \beta < 3$, their giant component is of size $\Theta(n) < n$ [41], [42]; and for $\beta \geqslant 3$ the graphs break up into components of order $o(n)$. Unfortunately, the leading constant of the size of the giant component is unknown. Further, a numerical estimation is hard to make since it is governed by a non-linear system of equations together with other parameters [20].

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 α: The parameter α adjusts the power law exponent β of the hyperbolic random graph via the functional behavior $\beta = 2\alpha + 1$ [12], [13]. The established way to estimate β is the algorithm by Clauset *et al.* [43]. Clauset *et al.* do not report an asymptotic runtime for their algorithm, but it appears to be superlinear. The practical runtime of freely available reference implementations, however, is dominated by our actual embedding algorithm. Moreover, Clauset *et al.* [43] also suggest an alternative algorithm, which is slightly less precise and runs in linear time. Preliminary experiments showed that the linear time method was actually very accurate, leading to no noticeable differences in the resulting embeddings. Thus, as the different methods do not show significant differences in quality or practical runtime, we opted for the freely available reference implementation in our experiments.

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 [33]. For small T, the fitness landscapes look virtually the same up to rescaling (see Figures 2a and 2b). In these cases, the attractive forces of neighbors dominate and the fitness is high close to their neighbors. We found that setting T to a small fixed value like 0.1 produces good results.

Increasing T emphasizes non-neighbors. The algorithm then places nodes in an area where there are few non-neighbors, while essentially disregarding the information from neighbors (see Figure 2d). Even though there is a short intermediate transition of the fitness landscape as can be seen in Figure 2c, our experiments suggested that setting T to a small value—even if the graph was generated using a large T—produced cleaner embeddings. For instance, Figure 3 contains the original vs. embedded angle of two embeddings where one has been computed using the original value of $T = 0.7$ and the other with $T = 0.1$. The algorithm performs better when using $T = 0.1$, even though the original T that has been used to generate the graph was large.

Estimating R and r_i: We estimate these values using the above determined parameters. Good analytical estimates have been derived in previous work [20]:

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

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

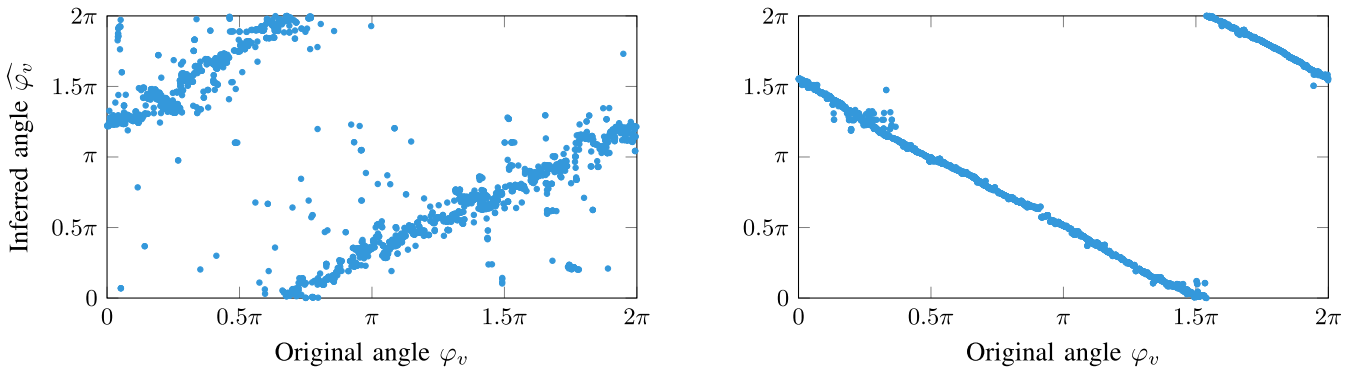

Fig. 6. The plots correspond to embeddings with average squared deviation $\Delta\varphi_G = 0.44$ (left) and $\Delta\varphi_G = 0.01$ (right). For each vertex v the plot contains one point with x-coordinate φ_v (angle of v in the original embedding) and y-coordinate $\widehat{\varphi_v}$ (angle in the computed embedding). The embedding is considered good if the plot resembles the identity function $f(x) = x$ up to cyclic shift and rotation.

B. Embedding the Core

Laying out large-degree nodes (also called the *core* of the graph) is critical for 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 $\Theta(n^{1-\alpha})$ with probability $1 - \mathcal{O}(\frac{1}{n})$ [44]. If these nodes have roughly the same circular ordering (in terms of their angular coordinates) in both the embedding and the generated graph, the remaining algorithm yields excellent embeddings. On the other hand, if the core was embedded poorly, the remaining steps cannot salvage this. We therefore put considerable care into embedding the core correctly.

HyperMap [34] uses the number of common neighbors of large degree nodes to infer their relative angles: For two nodes u, v they determine $c_{uv} = |\Gamma(u) \cap \Gamma(v)|$ and numerically compute the angle $\varphi(c_{uv}, r_u, r_v)$ that maximizes the likelihood that the nodes u, v have c_{uv} common neighbors. This approach is robust since the number of common neighbors of large degree nodes is tightly concentrated around its expected value. Determining the likelihood numerically, however, is a computationally expensive operation.

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

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. We have experimentally found, however, 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 $\Delta\varphi_{u,v}$. W.l.o.g., 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 [13] that this only holds if

$$\Delta\varphi_{u,w} \leqslant 2e^{\frac{1}{2}(R-r_u-r_w)}(1 + \Theta(e^{R-r_u-r_w})),$$

and

$$\Delta\varphi_{v,w} \leqslant 2e^{\frac{1}{2}(R-r_v-r_w)}(1 + \Theta(e^{R-r_v-r_w})). \tag{5}$$

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 in R and thus the inequality is satisfied for any angle $\Delta\varphi_{v,w}$ if R is sufficiently large. Thus, under the assumption that R is sufficiently large, we may use (5).

Observe now that for large enough radii r_w, the node w is not connected to either u or v (unless $\Delta\varphi_{u,v} \leqslant \mathcal{O}(\frac{1}{n})$). 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 $p_g(r_w)$.

As discussed, $p_g(r_w) = 1 \Leftrightarrow r_w = R - r_v \pm \Theta(1)$. We label this critical value of r_w with r_1. On the other hand, $p_g(r_w) = 0$ holds when $\theta(r_u, r_w) + \theta(r_v, r_w) \leqslant \Delta\varphi_{u,v}$, since then there is no possible angle for φ_w where it is connected to both nodes u, v, see (4). The critical value r_0 for which this number becomes positive is when $\theta(r_u, r_w) + \theta(r_v, r_w) = \Delta\varphi_{u,v}$ and thereby

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

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

For values $r_1 \leqslant r_w \leqslant r_0$, the regions in which w connects to u, v both increase as in (5). 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

$$1 = p_g(r_1) = A \cdot e^{-r_1/2} + B,$$

and

$$0 = p_g(r_0) = A \cdot e^{-r_0/2} + B.$$

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

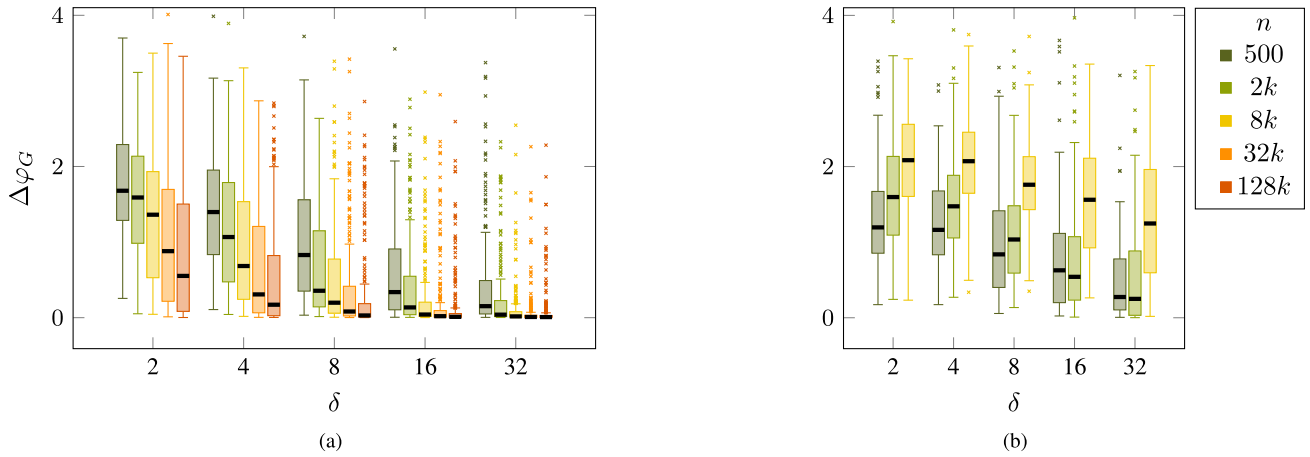

Fig. 7. Each data point in the box plot represents the value of $\Delta\varphi_G$ for a single graph G (y-axis) depending on the average degree a (x-axis). Smaller values are better. The graphs are grouped by their sizes. (a) Our main algorithm. (b) Our hyperbolic spring embedder.

v using the cumulative distribution function and p_g, and we have

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

Thus, the expected number of common neighbors of u, v is

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

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 $S \subseteq D_R$ in which each node is connected to both u, v and each node in $D_R \setminus S$ 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 $\Delta\varphi_{u,v}$ by rearranging the above equation.

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

To obtain actual values for $\Delta\varphi_{u,v}$ we first simply omit the constant factor hidden by $\Theta(1)$ in the above expression. Then, observe that the largest angle should likely be π. 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 π. 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 π after this scaling are then set to π. 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.

Embedding According to the Estimated Angles: In this section, we assume that we know the desired angle $\Delta\varphi_{u,v}$ 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 well as possible. To this end, we use a 1-dimensional spring embedder (see Section III for a short introduction to spring embedders) that 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 $0 \leqslant \varphi_u < \varphi_v \leqslant \pi$. Moreover, let $\mathrm{err}(u, v) = \varphi_v - \varphi_u - \varphi(c_{uv}, r_u, r_v)$. The force $F_u(v)$ acting on u due to v is then given by

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

To interpret this formula, note that $\mathrm{err}(u, v) < 0$ holds if the current angle is too small. Thus, $F_u(v)$ is negative (pushing u away from v) and it 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 decreases again, for the following reason. Imagine the extreme case that u and v have angle π between them but actually want to have a very small angle. Then it does not matter whether the angle of u increases or decreases as it comes closer to v. 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 $\mathrm{err}(u, v)$ becomes very large.

Similar to Section III, the total force 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 III: to move a vertex u to a specific position, 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

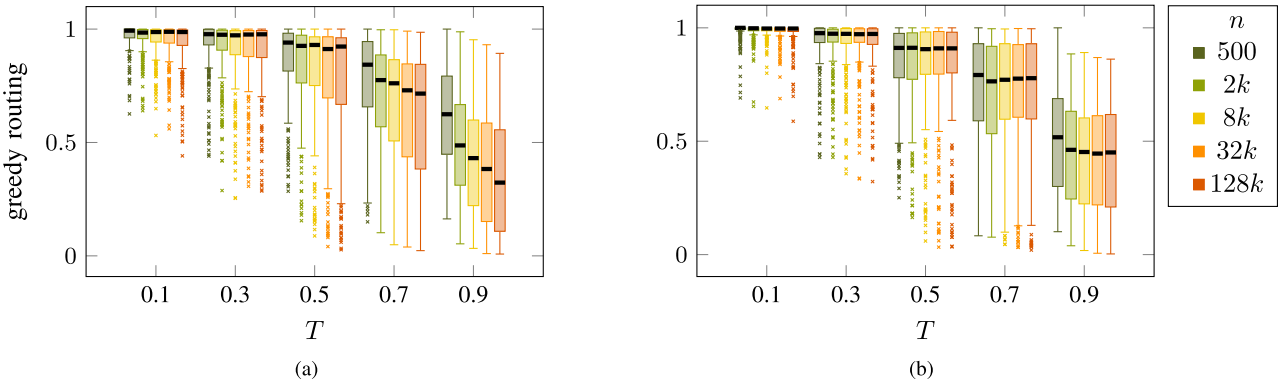

Fig. 8. The success ratio of greedy routing (x-axis) depending on the value of T (y-axis) grouped with respect to the number of vertices (colors). (a) Our main algorithm. (b) Originally generated embeddings.

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 [44], 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_{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 6 000 randomly generated hyperbolic random graphs (see Section VI 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).

C. Computing the Log-Likelihood Efficiently

A further key ingredient to achieve a quasilinear runtime is to improve the runtime of the Log-likelihood computation $\mathcal{L}(v)$. Recall that $\mathcal{L}(v)$ was defined as

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

see (3). By a naive implementation, one needs $\Omega(n)$ time to compute the Log-likelihood of a single node and thus at least $\Omega(n^2)$ for the whole graph. A more careful inspection, however, allows for a significant speedup.

First, observe that the total number of edges in a hyperbolic random graph is of order $\mathcal{O}(n)$ in expectation; 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

$$\begin{aligned} p_{uv} &= (1 + \exp(\frac{1}{2T}(\text{dist}(u,v) - R)))^{-1} \\ &\approx \exp(-\frac{1}{2T}(\text{dist}(u,v) - R)), \end{aligned}$$

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

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

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

$$p_{uv} \approx 1 - \exp(\frac{1}{2T}(\text{dist}(u,v) - R)) \to 1, \tag{6}$$

and thus

$$\begin{aligned} \log(1 - p_{uv}) &\approx \log(1 - (1 - \exp(\frac{1}{2T}(\text{dist}(u,v) - R)))) \\ &= \frac{1}{2T}(\text{dist}(u,v) - R). \end{aligned}$$

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.* [15]. These were originally used to generate hyperbolic random graphs in linear time by partitioning the disk D_R 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 significant influence on the Log-likelihood) while averaging over all nodes in far away cells. As shown in [15], 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.

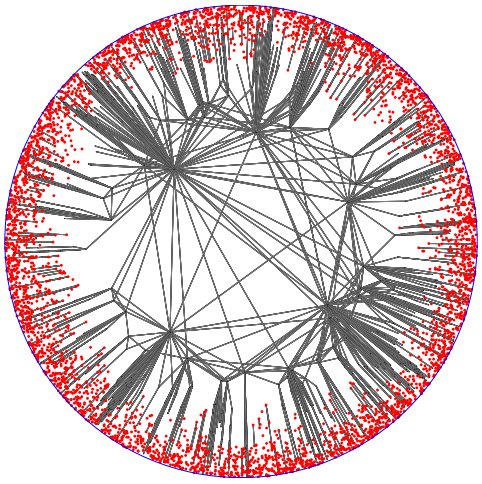

Fig. 9. When the average degree δ is small for the graph generation, most nodes (shown in red) are not part of the giant. It becomes hard to infer the original number of nodes based on the few that remain in the giant component.

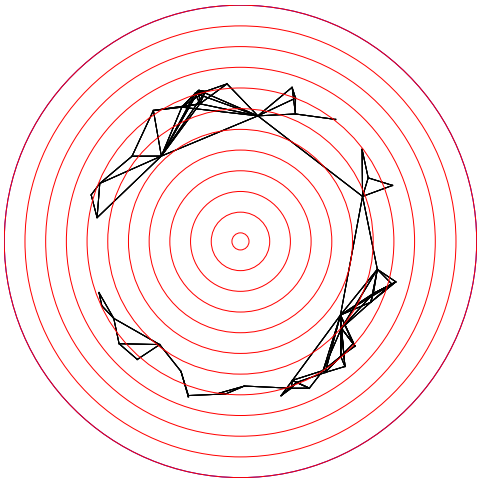

Fig. 10. When $\beta \to 3$, all nodes are pushed away from the center of D_R. The core thus attains a sparse, circular structure, for which the algorithm is not tailored.

Figure 5 shows the fitness landscapes of a node v computed once via the classical exact $\Omega(n)$ method, and once using our amortized $\mathcal{O}(\log n)$ method. Both methods exhibit no visible differences in the plot. Moreover, 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 $\leqslant 1.02$.

D. 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 its maximum likelihood. To determine this region, we observe that the coarse likelihood landscape for a node 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 4 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.

VI. Experiments

To evaluate the embedding quality, we sampled 10 hyperbolic random 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 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. A box contains 50% of all data points closest to the median, which is marked black. The size of the box is called interquartile range (IQR). Data points are considered outliers if they have a distance of more than $1.5\times$IQR to the box. The whiskers depict the closest data point to the box that is not an outlier.

A. Quality

A popular way to judge the quality of an embedding is to plot the embedded angular coordinates against the original generated coordinates. If the result resembles a straight line (up to 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 6. 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 φ_v in the original embedding and φ_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 7a plots $\Delta\varphi_G$ against the average degree δ; 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 δ, 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

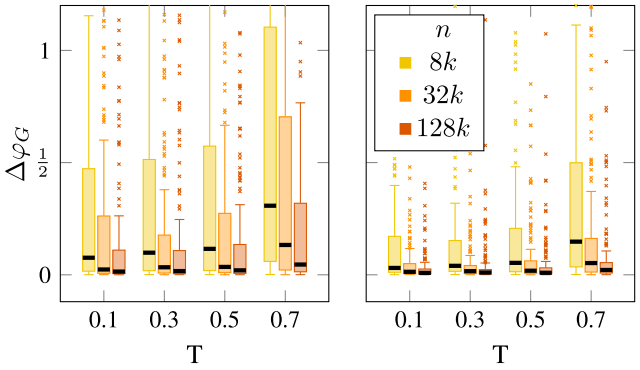

Fig. 11. Reevaluation of the experiments for embeddings, where hard cases $\delta = 2$ and $\beta = 2.9$ were discarded, see Section VI-B. This reveals that the embeddings of non-degenerate instances are of high quality, and most bad embeddings stem from hard corner cases. (a) Evaluation using all data. (b) Evaluation excluding degenerate cases.

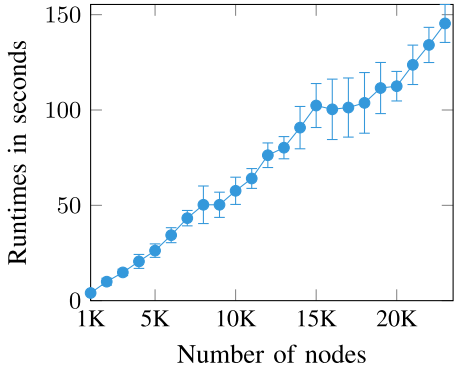

Fig. 12. Runtimes for the embedding algorithm. Error bars show the standard deviation.

for $\Delta\varphi_G$ of all medium sized ($n = 8\,000$) and large ($n = 128\,000$) graphs is 0.2 and 0.03, respectively. For comparison, note that the plots in Figure 6 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, see Figure 7b. As the spring embedder is too slow on larger graphs, we only ran experiments on graphs up to size $n = 8\,000$. Note that the quality of the spring embedder decreases for increasing graph size. In contrast, it performs comparatively well on small graphs 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 8a 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 the clustering coefficients of real-world networks are typically not exceedingly high, which corresponds to fairly large values of T in the model. For the embedding of the Internet graph [20], $T = 0.7$ was used. 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 8b.

Though maximizing the Log-likelihood leads to good success ratios, Figure 8b implies that even a perfect maximum likelihood embedder (one that always discovers the ground truth) cannot be expected to produce embeddings with 100% success ratio. Conversely, optimizing the embedding 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 non-perfect success ratios our embeddings achieve for large T as a weakness but rather as a strength, as this matches the behavior of the original embedding.

B. Further Work

Even though the algorithm produces meaningful embeddings overall, we observed that certain parameter combinations may lead to bad embeddings. In particular, this happens when (i) T is close to 1, (ii) δ is small or (iii) β is close to 3. Case (i) poses an inherent problem: If T was chosen large during the graph generation, random edges become more prevalent while the geometry plays a background role. Thus, it is natural that it is hard to embed these graphs meaningfully.

The other cases are less intuitive. In case (ii), the average degree is small. This leads to the generated graph having a small giant component. For instance, when $\delta = 2$, $\beta = 2.1$ and $T = 0.1$, a generated graph with $5\,000$ nodes only has 800 in its giant component (see Figure 9). Since only the giant is fed to the embedder, this results in a severe reduction of information. Consequently, the algorithm infers wrong parameters R, r_i which leads to a significantly different embedding than the ground truth. Note that when supplied correct values of R, r_i, the algorithm again produces embeddings of high quality. We are, however, not aware of a robust method that can infer these parameters in this degenerate case.

In the case (iii) when $\beta \to 3$, a different problem arises. Increasing β corresponds to shifting all nodes away from the center. Consequently, the core has a ring-like shape. Most high-degree nodes are then only connected to a few other nodes in the core. This situation results in few common neighbors (see Figure 10). As the core embedder in Section V-B relies on a dense matrix of common neighbor information, it fails to produce a good initialization which leads to a bad embedding. While rings can in principle be embedded well with classical spring embedders, these fail for dense cores. Thus, a refined core embedding algorithm that switches between these methods could improve upon the quality in this case. We argue, however, that this case is degenerate since such a ring-like structure most likely does not appear in the core of real-world graphs.

Figure 11 shows the performance of our algorithm on non-degenerate cases. On large graphs, our algorithm performs extremely well if the generation parameters are non-degenerate. This shows a clear road map on how the algorithm can be improved to achieve even better results overall.

C. Runtime

A key contribution of our algorithm is its significant improvement on runtime compared to previous approaches.

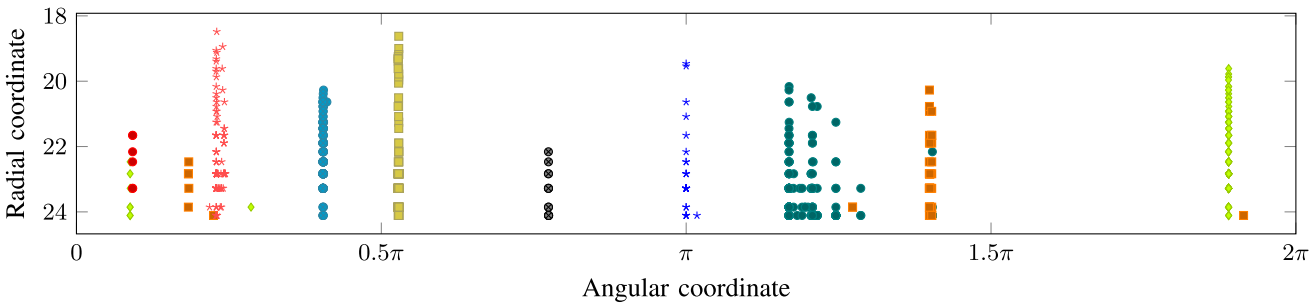

Fig. 13. The nine largest communities in the amazon product recommendation network. For clarity, only the 893 nodes that belong to a single community are shown. Nodes belonging to the same community are typically placed nearby, even though the embedding algorithm has no knowledge of the ground truth communities.

We performed runtime experiments on commodity hardware, i.e., a single 2.7 GHz Core i7 with 8 GB of RAM. Figure 12 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 VI-D. For comparison, the reference algorithm HyperMap [33], [34] needs over 1.5 hours for a graph of size 2 000.

D. Embedding a Real-World Graph

As a proof of concept, we embed the Amazon product recommendation network [45]. 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 13. Most nodes belonging to a single community are mapped close together. This suggests that the hyperbolic embedding might be a useful tool in discovering hidden communities in a large network.

VII. Conclusion

We designed and implemented a new algorithm for embedding complex networks into the hyperbolic plane. Connected nodes are typically placed close by, whereas disconnected nodes have a large hyperbolic distance. Compared to previous algorithms, we are the first to achieve a quasilinear runtime. This enables us to embed significantly larger graphs than before. Further, as we experimentally validated, our algorithm produces embeddings close to the ground truth; especially when either the number of nodes n or the average degree δ is large. In particular, the average angular error for embedded nodes becomes as small as 0.03 for $n = 128\,000$ and $\delta = 8$.

Our work was focused on presenting a proof of concept. As a benchmark, we used generated instances, which has two advantages. First, generated instances have a ground truth to which we can compare our embeddings. Second, they are easy to obtain in large quantities and thus allow for extensive experiments. The next logical step is to use our algorithm to embed real-world instances. This is interesting from an engineering perspective, as good performance on generated instances does not imply good performance on real-world instances (although considering random graphs with large T leads to a certain robustness). To conduct such experiments, one first needs to define a meaningful quality measure for embeddings of graphs without a ground truth, which is an interesting task in itself. Further, it is interesting to see how the embeddings can be used to learn new information about the behavior of real-world graphs. Hyperbolic embeddings were used before to produce efficient greedy routing [20], but other applications come to mind. For instance, a geographical representation of nodes opens new possibilities for finding clusters [46]. In fact, a different embedding algorithm reverses this idea by first computing clusters in the graph and then inferring node positions based on the found clusters [35].

A different direction is to use the embedding for visualization of massive networks. In fact, the hyperbolic plane was often used for visualization purposes [21], [23]–[25], [30]. Due to their size, classical methods typically struggle with finding a visual representation of the network that still conveys meaningful information. While the currently produced plots still only work for medium-size graphs before they become too cluttered, this may be improved by, e.g., (i) hiding "unimportant" edges as in [20], or (ii) providing a Focus+Context-like graph browser that allows for changing the coordinate origin as in [24] and [25]. Such tools magnify different regions of the graph while still placing the inspected nodes into the general graph context.

Finally, graph algorithms on hyperbolic random graphs that require knowledge of the geometrical representation can be invoked once we obtain the graph embedding. For instance, it has been shown that on hyperbolic random graphs, structures such as matchings and independent sets may be found more efficiently than on general graphs [47].

Acknowledgements

The authors thank Papadopoulos *et al.* [33] for their code and helpful discussions; C. Kessler and M. Katzmann (HPI Potsdam) for help with experiments; and K. Schöbel (FSU Jena) for help on hyperbolic variants of MDS.

References

[1] T. Bläsius, T. Friedrich, A. Krohmer, and S. Laue, "Efficient embedding of scale-free graphs in the hyperbolic plane," in *Proc. 24th Eur. Symp. Algorithms (ESA)*, 2016, pp. 16:1–16:18.

[2] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.
[3] R. van der Hofstad. (2016). *Random Graphs and Complex Networks*. [Online]. Available: http://www.win.tue.nl/~rhofstad/NotesRGCN.pdf
[4] F. Chung and L. Lu, "Connected components in random graphs with given expected degree sequences," *Ann. Combinatorics*, vol. 6, no. 2, pp. 125–145, 2002.
[5] W. Aiello, F. Chung, and L. Lu, "A random graph model for power law graphs," *Experim. Math.*, vol. 10, no. 1, pp. 53–66, 2001.
[6] W. Aiello, F. Chung, and L. Lu, "A random graph model for massive graphs," in *Proc. 32nd Symp. Theory Comput. (STOC)*, 2000, pp. 171–180.
[7] I. Norros and H. Reittu, "On a conditionally Poissonian graph process," *Adv. Appl. Probab.*, vol. 38, no. 1, pp. 59–75, 2006.
[8] A. Vázquez, "Growing network with local rules: Preferential attachment, clustering hierarchy, and degree correlations," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 67, no. 5, p. 056104, 2003.
[9] M. E. J. Newman, "Clustering and preferential attachment in growing networks," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 64, no. 2, p. 025102, 2001.
[10] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *J. Amer. Soc. Inf. Sci. Technol.*, vol. 58, no. 7, pp. 1019–1031, 2007.
[11] M. D. Penrose, *Random Geometric Graphs*. London, U.K.: Oxford Univ. Press, 2003.
[12] D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguñá, "Hyperbolic geometry of complex networks," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 82, no. 3, p. 036106, 2010.
[13] L. Gugelmann, K. Panagiotou, and U. Peter, "Random hyperbolic graphs: Degree sequence and clustering," in *Proc. 39th Int. Colloq. Automata, Lang. Programm. (ICALP)*, 2012, pp. 573–585.
[14] T. Friedrich and A. Krohmer, "On the diameter of hyperbolic random graphs," in *Proc. 42nd Int. Colloq. Automata, Lang. Programm. (ICALP)*, 2015, pp. 614–625.
[15] K. Bringmann, R. Keusch, and J. Lengler, "Geometric inhomogeneous random graphs," ETH Zürich, Zürich, Switzerland, Tech. Rep., 2015. [Online]. Available: https://arxiv.org/abs/1511.00576v2
[16] C. Koch and J. Lengler, "Bootstrap percolation on geometric inhomogeneous random graphs," in *Proc. 43rd Int. Colloq. Automata, Lang. Programm. (ICALP)*, 2016, pp. 147:1–147:15.
[17] R. Aldecoa, C. Orsini, and D. Krioukov, "Hyperbolic graph generator," *Comput. Phys. Commun.*, vol. 196, pp. 492–496, Nov. 2015.
[18] M. von Looz, H. Meyerhenke, and R. Prutkin, "Generating random hyperbolic graphs in subquadratic time," in *Proc. 26th Int. Symp. Algorithms Comput. (ISAAC)*, 2015, pp. 467–478.
[19] E. Stai, V. Karyotis, and S. Papavassiliou, "A hyperbolic space analytics framework for big network data and their applications," *IEEE Netw.*, vol. 30, no. 1, pp. 11–17, Jan./Feb. 2016.
[20] M. Boguñá, F. Papadopoulos, and D. Krioukov, "Sustaining the internet with hyperbolic mapping," *Nature Commun.*, vol. 1, Sep. 2010, Art. no. 62.
[21] J. A. Walter and H. Ritter, "On interactive visualization of high-dimensional data using the hyperbolic plane," in *Proc. 8th ACM Intl. Conf. Knowl. Discovery Data Mining (SIGKDD)*, 2002, pp. 123–132.
[22] J. A. Walter, "H-MDS: A new approach for interactive visualization with multidimensional scaling in the hyperbolic space," *Inf. Syst.*, vol. 29, no. 4, pp. 273–292, 2004.
[23] T. Munzner, "Exploring large graphs in 3D hyperbolic space," *IEEE Comput. Graph. Appl.*, vol. 18, no. 4, pp. 18–23, Jul. 1998.
[24] J. Lamping, R. Rao, and P. Pirolli, "A focus+context technique based on hyperbolic geometry for visualizing large hierarchies," in *Proc. 13th ACM Conf. Hum. Factors Comp. Syst. (CHI)*, 1995, pp. 401–408.
[25] J. Lamping and R. Rao, "The hyperbolic browser: A focus+context technique for visualizing large hierarchies," *J. Vis. Lang. Comput.*, vol. 7, no. 1, pp. 33–55, 1996.
[26] T. F. Cox and M. A. Cox, *Multidimensional Scaling*. Boca Raton, FL, USA: CRC Press, 2000.
[27] J. R. Clough and T. S. Evans, "Embedding graphs in lorentzian spacetime," *PLoS ONE*, vol. 12, no. 11, p. e0187301, 2016.
[28] D. M. Asta and C. R. Shalizi, "Geometric network comparisons," in *Proc. 31st Conf. Uncertainty Artif. Intell. (UAI)*, 2015, pp. 102–110.
[29] Y. Shavitt and T. Tankel, "Hyperbolic embedding of Internet graph for distance estimation and overlay construction," *IEEE/ACM Trans. Netw.*, vol. 16, no. 1, pp. 25–36, Feb. 2008.
[30] K. Verbeek and S. Suri, "Metric embedding, hyperbolic space, and social networks," in *Proc. 30th Symp. Comput. Geometry (SoCG)*, 2014, p. 501.
[31] X. Zhao, A. Sala, H. Zheng, and B. Y. Zhao, "Efficient shortest paths on massive social graphs," in *Proc. 7th Int. Conf. Collaborative Comput. (CollaborateCom)*, Oct. 2011, pp. 77–86.
[32] F. Papadopoulos, M. Kitsak, M. Á. Serrano, M. Boguñá, and D. Krioukov, "Popularity versus similarity in growing networks," *Nature*, vol. 489, no. 7417, pp. 537–540, 2012.
[33] F. Papadopoulos, C. Psomas, and D. Krioukov, "Network mapping by replaying hyperbolic growth," *IEEE/ACM Trans. Netw.*, vol. 23, no. 1, pp. 198–211, Feb. 2015.
[34] F. Papadopoulos, R. Aldecoa, and D. Krioukov, "Network geometry inference using common neighbors," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 92, no. 2, p. 022807, 2015.
[35] Z. Wang, Q. Li, F. Jin, W. Xiong, and Y. Wu, "Hyperbolic mapping of complex networks based on community information," *Phys. A, Statist. Mech. Appl.*, vol. 455, pp. 104–119, Aug. 2016.
[36] G. Alanis-Lobato, P. Mier, and M. A. Andrade-Navarro, "Efficient embedding of complex networks to hyperbolic space via their Laplacian," *Sci. Rep.*, vol. 6, Jul. 2016, Art. no. 30108.
[37] G. Alanis-Lobato, P. Mier, and M. A. Andrade-Navarro, "Manifold learning and maximum likelihood estimation for hyperbolic network embedding," *Appl. Netw. Sci.*, vol. 1, Dec. 2016, Art. no. 10.
[38] S. G. Kobourov, "Force-directed drawing algorithms," in *Handbook of Graph Drawing and Visualization*. Boca Raton, FL, USA: CRC Press, 2013, pp. 383–408.
[39] S. G. Kobourov and K. Wampler, "Non-Eeuclidean spring embedders," *IEEE Trans. Vis. Comput. Graph.*, vol. 11, no. 6, pp. 757–767, Nov./Dec. 2005.
[40] K. Q. Weinberger and L. K. Saul, "Unsupervised learning of image manifolds by semidefinite programming," *Int. J. Comput. Vis.*, vol. 70, no. 1, pp. 77–90, 2006.
[41] M. Bode, N. Fountoulakis, and T. Müller, "On the giant component of random hyperbolic graphs," in *Proc. 7th Eur. Conf. Combinatorics, Graph Theory Appl.*, 2013, pp. 425–429.
[42] M. Bode, N. Fountoulakis, and T. Müller. (2014). *The Probability That the Hyperbolic Random Graph is Connected*. [Online]. Available: www.math.uu.nl/~Muell001/Papers/BFM.pdf
[43] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," *SIAM Rev.*, vol. 51, no. 4, pp. 661–703, 2009.
[44] T. Friedrich and A. Krohmer, "Cliques in hyperbolic random graphs," in *Proc. 34th IEEE Conf. Comput. Commun. (INFOCOM)*, Apr./May 2015, pp. 1544–1552.
[45] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowl. Inf. Syst.*, vol. 42, no. 1, pp. 181–213, 2015.
[46] Z. Wang, Q. Li, W. Xiong, F. Jin, and Y. Wu, "Fast community detection based on sector edge aggregation metric model in hyperbolic space," *Phys. A, Statist. Mech. Appl.*, vol. 452, pp. 178–191, Jun. 2016.
[47] T. Bläsius, T. Friedrich, and A. Krohmer, "Hyperbolic random graphs: Separators and treewidth," in *Proc. 24th Eur. Symp. Algorithms (ESA)*, 2016, pp. 15:1–15:16.

Thomas Bläsius, photograph and biography not available at the time of publication.

Tobias Friedrich, photograph and biography not available at the time of publication.

Anton Krohmer, photograph and biography not available at the time of publication.

Sören Laue, photograph and biography not available at the time of publication.