

Random Minimum Spanning Trees in the Hyperbolic Plane

Bachelor Thesis of

Yavor Vasilev Lakov

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers: T.T. – Prof. Dr. Thomas Bläsius
PD Dr. Torsten Ueckerdt
Advisors: Maximilian Katzmann
Christopher Weyand

Time Period: 15th July 2022 – 15th November 2022

Statement of Authorship

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, November 14, 2022

Abstract

Networks are great at presenting the world that surrounds us in a way that is easy to observe. This is especially true for transmission trees, which can aid in understanding how diseases spread, but the medical data needed to generate them is scarce. Therefore, finding a way to simulate them is important. In this thesis, we propose to do this using spanning trees on points distributed randomly in the hyperbolic plane. To this end, we develop an algorithm for the generation of random hyperbolic minimum spanning trees. We develop an implementation of the algorithm and evaluate its efficiency and correctness, as well as investigate the properties of the resulting tree. Furthermore, we explore a case study of one hyperbolic minimum spanning tree and its node distribution appears to follow a power-law. Additionally, we have created a visual representation for the hyperbolic minimum spanning trees that result from this algorithm. For the testing of the correctness of the results, we generated 390 hyperbolic minimum spanning trees with diverse inputs of various sizes. According to our theoretical analysis, the complexity of the algorithm is in $O(n \cdot \log^2(n))$, which is confirmed by the empirical analysis. Additionally, we check how accurate the results of the hyperbolic distance function (which we used in our implementation of the algorithm) from the generator in [BFK⁺19] are, compared to a result computed with the highly accurate Boost library.

Deutsche Zusammenfassung

Netzwerke sind sehr gut darin, unsere Welt gut beobachtbar darzustellen. Dies gilt besonders für Übertragungsbäume, die helfen können, die Ausbreitung von Krankheiten zu verstehen. Leider sind die medizinischen Daten, die zu ihrer Generierung benötigt werden, knapp. Deswegen ist es wichtig, zu verstehen, wie wir sie simulieren können. Das Ziel dieser Bachelorarbeit ist es, einen Algorithmus zu entwickeln, der den minimalen Spannbaum auf einer gegebenen Anzahl von Punkten in der hyperbolischen Ebene berechnet. Wir entwickeln eine Implementierung des Algorithmus und bewerten seine Effizienz und Korrektheit. Wir untersuchen die Eigenschaften des resultierenden Baums. Darüber hinaus beobachten wir eine Fallstudie eines hyperbolischen minimalen Spannbaums und seine Knotenverteilung scheint einem “power-law” zu folgen. Außerdem haben wir eine visuelle Darstellung für die hyperbolischen minimalen Spannbäume erstellt, die sich aus diesem Algorithmus ergeben. Für die Überprüfung der Korrektheit der Ergebnisse haben wir 390 hyperbolische minimale Spannbäume mit diversen Eingaben unterschiedlicher Größe generiert. Gemäß unserer theoretischen Analyse liegt die Komplexität des Algorithmus in $O(n \cdot \log^2(n))$, was durch die empirische Analyse bestätigt wird. Zusätzlich überprüfen wir, wie genau die Ergebnisse aus der hyperbolischen Entfernung vom Generator (dessen Distanz-Funktion wir in unsere Implementation nutzen) in [BFK⁺19] im Vergleich zu einem Ergebnis sind, das mit der hochgenauen Boost-Bibliothek berechnet wurde.

Contents

1	Introduction	1
2	Related Work	3
3	Preliminaries	5
4	Random Hyperbolic MSTs	7
4.1	Basic Tools	7
4.1.1	Kruskal’s algorithm	7
4.1.2	Generator	7
4.2	Algorithm	8
5	Implementation	11
5.1	Implementation Notes	11
5.2	Testing	12
5.3	Graph Visualization	13
6	Experiments	17
6.1	Running Time Analysis	17
6.1.1	Theoretical analysis	17
6.1.2	Empirical analysis	18
6.2	Accuracy	22
6.3	Filtering	22
6.4	Degree Distrbution	25
7	Conclusion	27
	Bibliography	29

1. Introduction

Networks are an incredibly powerful tool for understanding the complicated world that surrounds us. Many processes and relationships can be represented by networks – viral diseases, roads, friendships, internet networks, romantic partnerships, and many others [BP16]. They all involve a vast number of entities and are astonishingly difficult to comprehend, especially in the form of raw data.

However, by using networks, we can transform that data into a legible form. If we want to extract any valuable information from such a process, we need to reduce the complexity of the information. Thus, in networks, we ignore all irrelevant information about each entity and view it simply as a node and we regard each connection between two entities as an edge between the two corresponding nodes. In the road example, nodes represent cities and an edge is an existing road between two cities.

Furthermore, what is great about real-world networks is that they share a lot of similarities. So whenever a new, unfamiliar network emerges, we can already use some of our knowledge about networks to extract approximate helpful information.

That can be very valuable when we need to understand how these processes work and we can make predictions about future developments by generating a simulation.

Simulations are especially helpful when it comes to the spreading of diseases. Correct modeling and simulation approaches are critical for elucidating the transmission of infectious diseases and improving the control of epidemics [JSC⁺21]. Keeping critical infrastructure in spite of an epidemic is important, but public health is a top priority, so good models can give a reasonable prediction of a situation and can save lives, as health officials can inform their decisions regarding adequate measures on such a model.

For example, the spreading of a disease can be illustrated via a *transmission tree*, which is a model that can help understand viral transmission, evaluate the effect of superspreading, and assess current preventative measures, as well as see evolutionary patterns of the disease. Unfortunately, as is the case with almost all real-world data, such data exists in only very limited availability, making it hard to analyze[YBS⁺12].

For that reason, there needs to be a way to simulate transmission trees and reap their benefits. There have been various approaches to generate trees; one of them are *random Euclidean minimum spanning trees (EMSTs)* [AESW91] (more about EMSTs in Section 3, which are the tree-counterpart of *random geometric graphs* [Pen03]). There is even an algorithm that efficiently calculates the minimum spanning trees of n points in the

Euclidean plane [CCK10]. The main shortcoming of EMSTs is their homogeneous degree distribution. This does not reflect reality, because superspreaders can spread disease to hundreds of people, in stark contrast to most of the infected, who usually spread it to a small number of people. More precisely, a “core” 20% of the host population usually contributes to more than 80% of transmission. This applies for various types of disease systems – from vector-borne parasites to sexually transmitted pathogens.[WDE⁺97]

To better reflect this heterogeneity, we propose to replace the underlying Euclidean geometry with hyperbolic geometry. In the case of random geometric graphs, this leads to *hyperbolic random graphs*, which feature a heterogeneous degree distribution [GPP12]. If we do the same with random EMSTs, the result is a *random hyperbolic minimum spanning tree*, which is the focus of this thesis.

Its main point is to explore an algorithm that finds a random hyperbolic minimum spanning tree of a number of randomly generated points on the hyperbolic plane. We offer an implementation of the algorithm, as well. The running time of the algorithm is in $O(n \cdot \log^2(n))$, which we prove both theoretically and through empirical data over 390 graphs with various parameters. Furthermore, our implementation finds the desired results in under 11 minutes for graphs consisting of up to 10^7 points. Additionally, we have created a visualization for the resulting minimum spanning trees that takes as input the coordinates of the random points, as well as a list of all edges, and returns an `.svg`-file that contains the minimum spanning tree, mapped onto the Euclidean plane in a legible way.

In Section 2, Related work, we review related work in the field that served as foundation for this thesis. In Section 3, Preliminaries, we define concepts that are necessary for understanding this thesis. In section 4, Random Hyperbolic MSTs, we define the concept of random hyperbolic MSTs, review basic tools, and explain the algorithm. In Section 5, Implementation, we discuss implementation details, explain the standard for testing the correctness of the implementation and results thereof, and show and discuss visual representations of the minimum spanning trees. In Section 6, Experiments, we discuss our theoretical expectations regarding the complexity of the algorithm and compare them with empirical data. Then, we compare the results from the hyperbolic distance function, available in `Point.h`, with the same formula, calculated with a higher accuracy with the help of `Boost`. Finally, we look at the degree distribution of a minimum spanning tree. Section 7 presents the conclusion of the thesis.

2. Related Work

This thesis builds on previous works on network science, hyperbolic random graphs and minimum spanning trees.

One of the foundational books about network science, Network Science Book by Albert-László Barabási [BP16] describes the universality of networks, regarding their ability to present relations from our world. In this book, the author discusses what the qualities are that a network model that can simulate real-world processes has to exhibit. Among the most important qualities is that the degree distribution of a graph should follow a *power-law*, meaning that the probability for a node to have a degree k is well approximated by $p_k \sim k^{-\gamma}$. In a good model of the real world, there have to be “hubs” – nodes that are extremely highly connected, but there is a very small number of them – as opposed to the vast majority of other nodes, which have a really small number of connections.

There have been many attempts to create a model that has such qualities or that realistically replicates actual networks. One such model is the Erdős–Rényi model [ER59]. In it, there is the same probability that each two nodes have an edge between them, regardless of the properties of each node or their relation to each other. Furthermore, random geometric graphs move that concept on the Euclidean plane [Pen03]. Such a graph consists of n randomly placed points and a threshold distance $r > 0$, so that an edge exists between two nodes, if and only if the norm of their difference is at most r . For trees, a way to model that is through EMSTs, that we introduce in Section 3. The problem with these models is that, as nodes are scattered on the Euclidean plane at random, the expected degree of all nodes is the same. Thus, by using these models, it is not possible to generate a graph, whose degree distribution follows a power-law and, therefore, the capabilities of these models to simulate the real world are limited.

In 2010, Krioukov et al. [KPK⁺10] introduced a new model, that was later given the name *hyperbolic random graph*, that this thesis heavily relies on. This model is described in Section 3. In the paper where the model was first introduced, the authors already show some of the properties that make it interesting, including strong heterogeneity and clustering.

After that, Gugelmann et al. [GPP12] further explore the hyperbolic random graph model. They prove other qualities of hyperbolic random graphs that make them an even more appealing model. They find that there is a constant lower bound on the clustering coefficient of hyperbolic random graphs, thus, confirming that in HRGs there is high clustering. Also,

the expected degree distribution follows a power-law up to the maximum degree and yields small probabilities for large deviations. All of these findings make HRGs very promising.

After it was established that HRGs are worth examining, Bläsius et al. [BFK⁺19] have built a generator that, given parameters of the model, returns a hyperbolic random graph. In addition, the generation of large graphs is very efficient, as a graph with 10^7 edges is generated in under one second. This generator is used in each step of the algorithm that we built. Now that HRGs can be easily generated, it is easier to observe them, to extract valuable information about them and to perform operations on them.

Another topic that is essential for this thesis are minimum spanning trees and, in particular, Kruskal’s algorithm [Kru56] for finding the minimum spanning tree of a graph. Minimum spanning trees as a concept are defined in Section 3 and an explanation of Kruskal’s algorithm is in Section 4.1.1. We use Kruskal’s algorithm as a basis to define our algorithm that finds a hyperbolic minimum spanning tree. However, using an unmodified version of the algorithm is not efficient, as its complexity, $O(m \cdot \log(m))$ [HP08], heavily depends on the number of edges, which in our case is in $O(n^2)$. Thus, the complexity of the algorithm is $O(n^2 \cdot \log(n))$. Nevertheless, if we modify the algorithm, a significant improvement can be achieved. Other minimum spanning tree algorithms exist, such as Prim’s [Pri57] and Borůvka’s [NMN01] algorithms, but neither of them are efficient for our case.

3. Preliminaries

We define a *graph* G as a pair (V, E) , consisting of a set of vertices and a set of edges. All graphs that we observe in this thesis are undirected and weighted. The number of nodes in a graph is denoted by n and the number of edges – m .

A *spanning tree* of a graph $G = (V, E)$ is a tree $T = (V, E')$, where the set of nodes is the same, but the edge set E' is such subset of E , so that a path exists between each two nodes and no cycles are present. A *minimum spanning tree* is a spanning tree with the minimal possible sum of weights among all spanning trees.

The concept of a hyperbolic minimum spanning tree is related to the concept of an *Euclidean minimum spanning tree*, so we define that first. For n points $V_E = \{v_1, v_2, \dots, v_n\}$ in Euclidean d -dimensional space $\mathbb{E}^d$, a *Euclidean minimum spanning tree (EMST)* is an MST of the complete graph $G_E = (V_E, E_E)$ of these n points, where $E_E = \{v_p v_q | 1 \leq i \neq j \leq n\}$ and the weight of each edge $v_p v_q$ is the Euclidean distance (see 3.1 – d is the dimension of the Euclidean space and v^i is the coordinate of the node v in the i -th dimension) between v_p and v_q [AESW91].

$$d_E(v_p, v_q) = \sqrt{\sum_{i=1}^d (v_p^i - v_q^i)^2} \quad (3.1)$$

The *hyperbolic geometry* is a non-Euclidean geometry. It has numerous interesting properties – for example, for the angles α, β and γ of a triangle, the following is true: $\alpha + \beta + \gamma < \pi$. It has a constant negative sectional curvature (-1) [VS16] and space expands exponentially [KPK⁺10]. We use *polar coordinates* to represent points on the hyperbolic plane. The polar coordinates of a point are a pair (d, θ) , where d is the distance from the point to the origin and θ is the angle between an axis and the segment connecting the point to the origin. We define the distance between two points p_1 and p_2 with their respective polar coordinates (r_1, ϕ_1) and (r_2, ϕ_2) , on the hyperbolic plane in Equation 3.2.

$$d_H(p, q) = \text{acosh}(\max(1.0, \cosh(r_1, r_2) + (1.0 - \cos(\phi_1 - \phi_2)) \cdot \sinh(r_1) \cdot \sinh(r_2))) \quad (3.2)$$

A *hyperbolic random graph (HRG)* is generated by distributing n points on the hyperbolic plane $\mathbb{H}^2$, uniformly at random over a disk of radius R . We assign the angular coordinate $\theta \in [0, 2\pi]$ of a node with the following density function: $\rho = 1/(2\pi)$. The density for the radial coordinate $r \in [0, R]$ has an exponential density function: (see Equation 3.3) [KPK⁺10].

$$\rho(r) = \frac{\alpha \cdot \sinh(\alpha r)}{\cosh(\alpha R - 1)} \approx e^{\alpha(r-R)} \quad (3.3)$$

Each two nodes have an edge between each other according to a function that depends on the hyperbolic distance between them. As the distance between two nodes grows, the probability with which they are connected shrinks proportionally. In the *threshold* variant, any two vertices are connected by an edge if their hyperbolic distance is at most R . Hyperbolic random graphs exhibit high clustering and have a power-law degree sequence up to the last degree. This makes HRGs a strong candidate for modeling real-world networks [GPP12].

Kruskal’s algorithm uses the *union-find* data structure to keep track of which nodes are in the same component. In the beginning, it consists of n separate entities and each of them is in its own component. We can merge any two components, meaning that all entities which belong to both of them are now part of the same component. It is also possible to make queries whether two entities belong to the same component.

4. Random Hyperbolic MSTs

In the following, we establish the most important concepts of the thesis: Hyperbolic minimum spanning trees, random hyperbolic minimum spanning trees, Kruskal’s algorithm and the generator for random hyperbolic graphs from [BFK⁺19].

A *hyperbolic minimum spanning tree (HMST)* of n points on the hyperbolic plane is a minimum spanning tree of the graph $G_H = (V_H, E_H)$, where V_H consists of the n points on the hyperbolic plane and E_H consists of all possible pairs of points, with the weight of the edge $v_i v_j \in E_H$ being the hyperbolic distance between v_i and v_j (see Equation 3.2). If the n points are sampled as for hyperbolic random graphs, which are already described in Section 3, we call this structure a *random hyperbolic minimum spanning tree (random HMST)*.

Before describing our algorithm for generating random HMSTs, we introduce the basic tools that we utilize during the generation – Kruskal’s algorithm and the generator of hyperbolic random graphs from [BFK⁺19].

4.1 Basic Tools

4.1.1 Kruskal’s algorithm

Kruskal’s algorithm finds a minimum spanning tree on a weighted graph. The algorithm has two steps. In the first step, we sort all edges, according to their weight. We call this order of the edges S . In the second step, we iterate over all edges in the order S , one by one. Initially, each node is in its own component and there are n components. If the two endpoints of the current edge are in the same component, we ignore it and carry on with the next one. Otherwise, we unite the two components, to which the two endpoints of the edge belong and, after that, we move to the next edge in S . Kruskal’s algorithm has a complexity of $O(m \cdot \log(m))$ [HP08].

4.1.2 Generator

We can generate a HRG in expected linear time with the help of the generator in [BFK⁺19]. The generator generates n nodes on the hyperbolic plane. One of the key characteristics of a HRG is R , the radius of the disk around each node, so that it has an edge to each of the other nodes within that radius. How high should R be? The value for R depends on the number of nodes, n , the average degree, d (usually constant), and the constant α for adjusting the power-law exponent. The generator receives these as input parameters and it calculates the desired value of R .

4.2 Algorithm

The input is the set of parameters n, α . We use the generator [BFK⁺19] to generate the points. What the algorithm does is that it finds the edges of the HMST. A naïve approach is to compute all edges, but, as we point out above, that is inefficient. Instead, we use the generator to compute the edges. For that, we need to specify an average degree. Since we do not know what average degree to choose, we start with a constant: 8. Stepwise, we double the average degree, use the generator for edges, filter for new ones and merge components like Kruskal until only one component is left.

Here is an outline of the algorithm. The edges are not considered separately – they are considered in “bucket sort”-style groups. Instead of generating all $\Theta(n \cdot \log(n))$ edges at once, we can start by generating a linear number of edges. From there, we can build a minimum spanning tree, using Kruskal’s algorithm stepwise on each of the buckets, and we can rely on the fact that the minimum spanning tree is close to connected in each step. We observe each edge e one by one and, using a union-find data structure, check whether the two nodes on both ends of e are connected (e.g. lie in the same union-find-component). If they are not in the same component, the edge is marked for later consideration. After all edges have been iterated, the edges that connect two different components are sorted according to the edge length. Then, we iterate over the sorted edges and each time, if the current edge connects two components, then it is added to the MST and the components are united, otherwise it is ignored. After all edges have been considered, if the graph is still not connected, then we double the number of edges, by doubling the input average degree d and the process is repeated. Algorithm 4.1 shows this procedure in pseudo-code.

Algorithm 4.1: GENERATING A RANDOM HMST

Input: Number of nodes n , power-law degree distribution control parameter α **Data:** Initial radius of the plane R , sample radii, angles, and edges from generator
– radii, angles, hrgEdges**Output:** Random hyperbolic MST – mst

```

// Initialization
1 deg  $\leftarrow$  8
2  $R_0 \leftarrow R$ 
3 numComponents  $\leftarrow$  n
4 iterations  $\leftarrow$  0
// Main loop
5 while numComponents > 1 do
6   if deg > n - 1 then
7     break
8    $R \leftarrow R_0 + 2 \cdot \text{iterations}$ 
9   recompute hrgEdges with the new R
10  forConsideration: vector of edges
11  forall  $(u, v, d) \in \text{hrgEdges}$  do
12    if FIND( $u$ )  $\neq$  FIND( $v$ ) then
13      forConsideration.PUSHBACK( $\{u, v, d\}$ )
14  forConsideration.SORT
15  forall  $(u, v, d) \in \text{forConsideration}$  do
16    if FIND( $u$ )  $\neq$  FIND( $v$ ) then
17      UNION ( $u, v$ )
18      mst.PUSHBACK( $\{u, v, d\}$ )
19  deg = deg  $\cdot$  2
20  iterations = iterations + 1
21
22 return mst

```

5. Implementation

5.1 Implementation Notes

We wrote the generator of random hyperbolic minimum spanning trees in `C++` because the code for generating radii [BFK⁺19], angles and edges was already in `C++` and we could use it directly, leaving less space for errors. Our code is available at https://git.scc.kit.edu/scale/research/hyperbolic-mst/-/blob/master/source/examples/mst_generator/main.cpp

The maximum possible number of nodes, defined in the code, is 10^7 , as the experiments were performed for sets of points with between 100 and 10^7 elements. The program reads the input as arguments from the console. The output is returned in four text files. One of them includes the results from the running time experiments along with data that can help identify the graph and additional information. The second output file consists of the hyperbolic MST that the program generated: the file starts with the number of nodes on top, then, there is a list of the polar coordinates of each node and, after that, there is a list of all edges in the hyperbolic MST. The third file includes data regarding each iteration of the algorithm – how many edges did the program generate and how many of them became part of the MST. The fourth file gives information regarding the correctness of the generated tree – whether the number of edges in the MST is $n - 1$, whether it is connected and, if the graph has 1000 nodes or less, whether the tree is minimal. The program saves all output files except the one which stores the resulting graph with the `fstream::app` flag, which does not erase any data already stored in the file where the program writes. That allows us to run a large number of tests through a script and, then, to process the data as a whole. If any of the outputs is not needed, they can be easily commented. The program saves the output in the top-level directory of the project.

The code uses the following libraries from the Standard Template Library (STL):

- `<fstream>`: used for writing the output into a text file
- `<algorithm>`: used for sorting chosen edges on each iteration of the algorithm, as well as for the `max` function.
- `<vector>`: used for various purposes throughout the code, including as part of the graph representation, for keeping the minimum spanning tree, for saving information regarding the behavior of the algorithm on each iteration (the number of iterations is not clear initially), and for saving edges that connect two different components for

later inspection (their exact number is also not known before the loop inspects all edges).

- `<time.h>`: used for measuring the running time of the relevant parts of the implementation (input, output and initialization of the point set are not included)
- `<tuple>`: used for the `tie` function that compares two tuples – it is useful for comparing edges

From the generator [BFK⁺19], we use the functions for calculating the initial radius R , for sampling radii and angles, and for generating edges. Also, we use the function in `Point.h` to calculate hyperbolic distances. In Section 6.2, we test the accuracy of that function. We have defined a structure for edges that includes the two endpoints, the distance, and a `<` operator that is used by the `sort` function in STL. Additionally, we have implemented a union-find structure, in order to distinguish whether two nodes are in the same component and to unite components. In addition, we keep track of the total number of components. This is necessary to determine the end of the algorithm: when the total number of components reaches 1, that means that all nodes are in the same component and we have found the hyperbolic MST.

To run the program, one should open the folder through the command prompt and write the following in the console:

```
./mst_generator <name> <rseed> <aseed> <sseed> <n> <alpha>
```

The parameters are as follows: `<name>` is the name of the test that is about to be performed – it can be any string (as a large number of experiments can be performed automatically, it is useful to be able to identify a particular graph that yields unexpected results – also, if the code for saving the graph representation of the hyperbolic MST is left uncommented, the program adds this name at the end of the file name to help find the correct file), `<rseed>`, `<aseed>`, and `<sseed>` are the seeds needed for the random generation of radii, angles and edges, respectively, `<n>` is the number of nodes on the plane, and `<alpha>` is a parameter needed for generating radii.

5.2 Testing

Testing whether the implementation returns an actual minimum spanning tree of the complete graph of the random set of points is crucial. For a tree T to be a minimum spanning tree, per definition, it has to be both spanning and minimal, meaning that no edge $uv \in E$ exists between two nodes on the tree whose inclusion would help make a tree with an even smaller sum of edge weights, by removing the largest edge on P , the path between u and v , and adding a new edge, uv , to the tree. Therefore, on each such path the highest weighted edge (the largest distance between two adjacent nodes on P) is smaller than the distance between u and v .

After generating 390 graphs with various values for α and n , tests for connectivity of the generated graphs were performed. If a resulting tree has two or more components, that suggests that the algorithm or the implementation is incorrect. To prove the connectivity of each graph, we executed a depth-first search, with the 0-th node as a starting point. According to the test, all 390 graphs were connected, which suggests that the algorithm and its implementation return a connected graph. This result affirms the correctness of the algorithm and its implementation.

Also, on all 130 graphs that have a number of edges less than or equal to 1000, a test for the minimality of the graph was performed. (For larger graphs, testing the minimality is significantly more difficult, as the total number of possible edges is in $\Theta(n^2)$ and, thus, the test is very slow. We assume that the same findings as for $n \leq 1000$ hold for larger values

of n as well.) For a pair of nodes, u and v , the path P between them on the spanning tree is observed. If the edge with the highest weight (weights represent the distances between the nodes) on P is larger than the hyperbolic distance between the two nodes, then there is a spanning tree with a smaller sum of the edge weights. In that case, the spanning tree cannot be minimal. All of the trees that were tested were minimal, according to this metric. These tests prove that the generated tree is, in fact, a minimum spanning tree in these 130 cases. As they are a diverse sample, this further suggests that the output of the algorithm/implementation is correct.

Furthermore, all MSTs, resulting from the implementation, have the correct number of edges that a spanning tree should have, namely $n - 1$, where n is the number of nodes in the graph.

As minimality, connectedness, and the correct number of edges are necessary and sufficient conditions for a spanning graph to be an MST [HP08], that proves that all 390 graphs, resulting from the tests are, in fact, hyperbolic minimum spanning trees of their respective point sets.

5.3 Graph Visualization

It is beneficial to be able to generate legible visualizations of the minimum spanning trees, generated by the algorithm, for two reasons. First, a visual model is a perceptible way of representing hyperbolic MSTs, which can help in observing interesting unexpected properties of the graph. The process for calculating the minimum spanning tree and testing it involve a massive amount of data (a `.txt` file describing one of the larger graphs that we generated often exceeds 300 MB), which is indecipherable to anyone trying to understand what the algorithm is doing and, consequently, we can gain very little information that way. Second, a visualization can serve as a sanity check for the implementation. It is another option for testing whether the behavior of the algorithm matches what is expected.

To achieve that, we need to present a hyperbolic MST on a screen, so we have to map hyperbolic coordinates to Euclidean ones. As hyperbolic space differs vastly from Euclidean space, the periphery of a Euclidean visualization is represented much denser than the center. Due to the large number of nodes in these dense regions, it is difficult to see anything there and to discern any valuable information regarding the graph. For that reason, it is important to find a way to present the graph without making the borders difficult to read.

In order to achieve that, we implemented a program, which receives a `.txt`-file, containing the polar coordinates of the points on the hyperbolic plane, as well as a list of all pairs of nodes that are connected by an edge. The program is implemented in `C++` and it returns a visualization of the graph as an `.svg`-file. As the visualization has to be on a simple, two-dimensional Euclidean plane, we used the function, defined in 5.1

$$h : (\mathbb{R}, [0, 2\pi)) \longrightarrow \mathbb{R}^2, (R, \varphi) \longmapsto (R \cdot \cos(\varphi) \cdot 100, R \cdot \sin(\varphi) \cdot 100) \quad (5.1)$$

to convert the polar coordinates into two-dimensional Cartesian ones. The program translates all coordinates with h and connects each neighboring pair with a line.

As most of the nodes on the tree are concentrated towards the periphery of the plane, the vast majority of the edges are difficult to read, in case they are too thick – especially for graphs with a large number of nodes. However, if the thickness or the opacity of the edges is lowered, then the edges of the few nodes closer to the middle, are hardly visible. Thus, a better solution is to find formulas for the thickness and opacity of the edges that depends on the distance to the origin.

For the thickness, we need a formula that accounts for the power-law and a good option proved to be a thickness function of the $f(x) = \frac{a}{b+x} + c$ type, where a , b and c are parameters and x is the distance. Through testing with various values, 10 and 0.1 proved to be adequate widths at the origin and at the periphery, respectively. Thus, the points $(0, 10)$ and $(R, 0.1)$ should lie on the graph of the function $f(x)$.

Let $b = R$. Solving the linear system of 2 equations leaves $a = 19.8R$ and $c = -9.8$. The resulting function f is in equation 5.2.

$$f(x) = \frac{19.8R}{R+x} - 9.8 = \frac{10R - 9.8x}{R+x} \quad (5.2)$$

For the opacity formula, it is enough to have the edges fully solid (opacity = 1.0) on the origin of the plane and to have opacity gradually decrease until an edge is half-transparent (opacity = 0.5) at distance R to the origin. The formula that we used is in 5.3.

$$g(x) = \frac{1 + \sqrt{1 - \frac{x}{R}}}{2} \quad (5.3)$$

Another problem around visualizations of the minimum spanning trees is that, as a consequence of the higher concentration of nodes towards the margins of the plane, they all appear as one large mass and, again, make it difficult to recognize any significant properties of the graph. For that reason, we implemented a formula, where the radius of the circles representing nodes decreases depending on the distance of a node to the origin.

With these formulas, we generated the visualization of a graph with $n = 100000$ (see Figure 5.1)

If the results from the visualization confirm our expectations, that further establishes the correctness of our algorithm and implementation. As expected, there are hardly any (only two) nodes very close to the origin and they are very highly connected. Also, there are a few nodes that are slightly further away but still have a high degree. The rest is the vast majority of the nodes that are on the periphery of the plane and have very few neighbors (rarely more than one – most of these nodes are leaves). Although there are 100 000 nodes, it is already visible that the structure that unites them is a tree, confirming our expectation and further proving the validity of the algorithm. Also, the entire graph consists of one component, which further supports the algorithm and the implementation.

We can observe the same by visualizing smaller trees, too (see Fig. 5.2). Here, due to the fringes of the plane being significantly less concentrated, it is even clearer that all nodes are connected and that the result of the algorithm is a tree. However, due to the properties of the hyperbolic plane, the minimality of the hyperbolic MSTs cannot be observed in the utilized projection to the Euclidean plane.

Of course, a visual test alone is not a complete proof of the correctness of an implementation or an algorithm. It only works as such in conjunction with other more thorough tests of the entire data set, which we also performed, as explained in Section 5.2.

This visualization is especially useful if the algorithm or its implementation are modified or repurposed in the future – if a new property of hyperbolic MSTs becomes important in the future, individual graph visualizations can be observed to help researchers intuitively determine boundaries for what is possible.

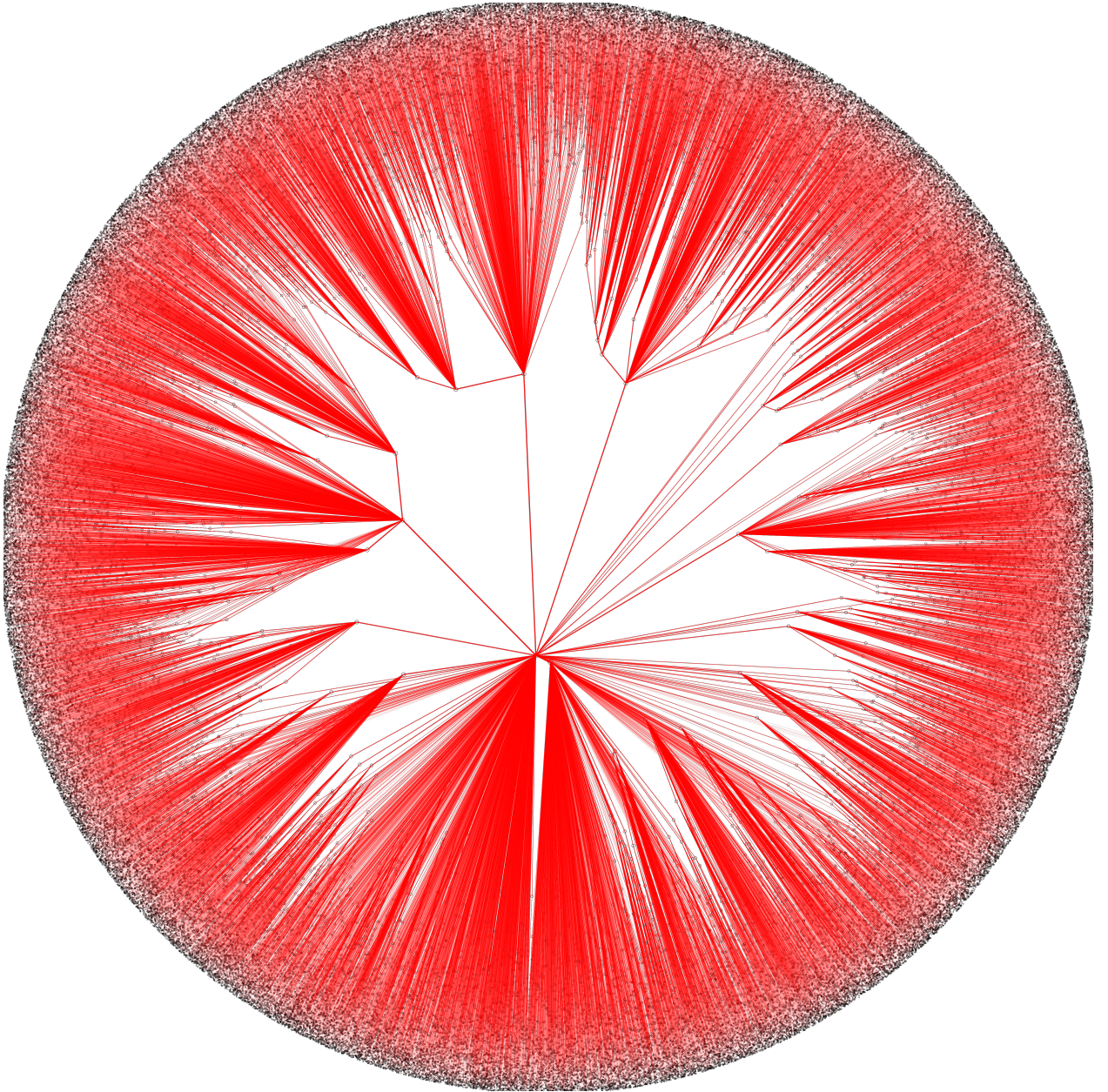


Figure 5.1: This is the minimum spanning tree of a random set of points with: $n = 100000$ and $\alpha = 0.6$. The seeds for generating the points are as follows: rseed = 2040, aseed = 3961, sseed = 4728. The points were generated in 5.813 s. The visualization was generated in 8.553 s.

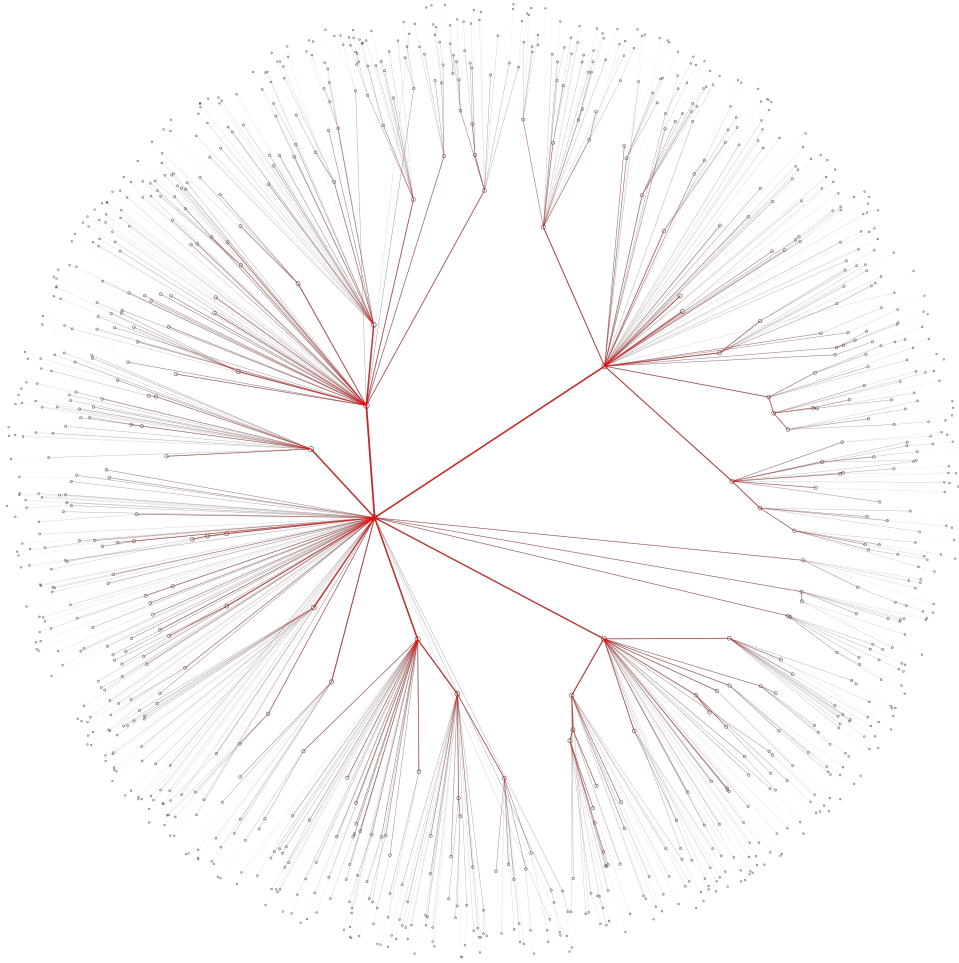


Figure 5.2: This is the minimum spanning tree of a random set of points with: $n = 1000$ and $\alpha = 0.6$. The seeds are the same as in the bigger tree. The points were generated in less than 0.001 s. The visualization was generated in 0.182 s.

6. Experiments

6.1 Running Time Analysis

6.1.1 Theoretical analysis

Kruskal's algorithm is quite efficient for graphs where the number of edges is relatively low. However, for a set of random nodes, we can draw a line between any two arbitrary nodes. That leaves the graph with $\Theta(n^2)$ edges and, thus, Kruskal will not perform well. Due to Kruskal's complexity being $O(E \cdot \log(|E|))$ [HP08], the complexity is $O(n^2 \log(n))$ in this case, which is not viable for samples where n is very large.

The initially generated graph is typically not connected. The number of nodes in the biggest component is in $\Theta(n)$. All other components are significantly smaller; they have size in $O(\log(n^{\frac{1}{1-\alpha}}))$ for $\alpha \in (0.5, 1)$. As the generation of a HRG requires an average degree input, it is important to answer the question, what the value of the average degree should be. A good average degree should not be too high. If the average degree is more than $\log(n)$, then there will be more than $n \cdot \log(n)$ edges. As any minimum spanning tree algorithm should iterate over all edges, that alone would bring the complexity of the whole algorithm over $n \cdot \log(n)$. Therefore, if the generated graph has a number of edges that is in $\omega(n \cdot \log(n))$, the algorithm is not efficient. However, the average degree should be just enough, so that the graph is connected. It is suspected that an average degree of $\Theta(\log(n))$ is the threshold from which a HRG is connected asymptotically almost surely. This is not certain yet, so it should be further investigated. It is possible that fewer edges are sufficient. However, if $\Theta(\log(n))$ really is the threshold, then an average degree of $\Theta(\log(n))$ would be good enough, so that the graph is connected, but the total number of edges is still in $O(n \cdot \log(n))$.

The complexity of the algorithm is not optimal – $O(n \cdot \log^2(n))$. Another algorithm exists that solves the problem in $O(n \cdot \log(n))$. Nevertheless, ours is numerically more stable, which is often important in practice.

The generator in [BFK⁺19] runs in expected linear time, we start with m edges and, in each step, the number of edges doubles. That means that, on the k -th iteration, there are $\Theta(m \cdot 2^k)$ edges that are considered. To find the number of iterations, we should find how large k can be. The graph is almost surely connected when $R' \approx R + 2 \cdot \log(\log(n))$ (we know that from preliminary computations). Also, we double the average degree by picking $R' = R + 2 \cdot \log(2)$. Therefore, in the k -th iteration, the radius is $R' = R + k \cdot 2 \cdot \log(2)$. Then,

$k \cdot \log(2) \in \Theta(\log(\log(n)))$, which means that the number of iterations is in $\Theta(\log(\log(n)))$. Therefore, the total number of edges inspected sums up to $O(n \cdot \log(n))$. So, the deciding factor is the sum of the sorting operations ($\Theta(m \cdot \log(m))$ per iteration) that happens on each iteration. In the i -th iteration, $m_i = m \cdot 2^i$, but we have $m \in \Theta(n)$. So, over all iterations, we have the sum in

$$\Theta\left(\sum_{i=1}^k m_i \cdot \log(m_i)\right) = \Theta\left(\sum_{i=1}^k m \cdot 2^i \cdot \log(m \cdot 2^i)\right) \quad (6.1)$$

$$= \Theta\left(\sum_{i=1}^k n \cdot 2^i \cdot \log(n \cdot 2^i)\right) \quad (6.2)$$

$$= \Theta\left(n \cdot \sum_{i=1}^k 2^i \cdot (\log(n) + i)\right) \quad (6.3)$$

$$\leq \Theta\left(n \cdot \sum_{i=1}^k 2^i \cdot (\log(n) + \log(\log(n)))\right) \quad (6.4)$$

$$= \Theta\left(n \cdot \log(n) \cdot \sum_{i=1}^k 2^i\right) \quad (6.5)$$

$$= \Theta\left(n \cdot \log(n) \cdot 2^{\log(\log(n))}\right) \quad (6.6)$$

$$= \Theta(n \cdot \log^2(n)) \quad (6.7)$$

Then, the complexity of the algorithm is in $\Theta(n \cdot \log^2(n))$.

6.1.2 Empirical analysis

We conducted 390 experiments to see how the running time of the implementation develops as n grows. All tests were performed on a x64-based PC with Microsoft Windows 11 Home, version 10.0.22000 Build 22000. The RAM of the machine is 8 GB. The processor has 2 cores and 4 threads at 2.20 GHz, 2201 Mhz.

The box plots in Figures 6.1 and 6.2 present the running time of the implementation on diverse inputs, where the values of α are 0.6, 0.7, 0.75, 0.8 and 0.9. The values of the seeds and the number of nodes vary. For all experiments, it takes less than 11 minutes to find a minimum spanning tree even for $n = 10^7$. The running time decreases with increasing α .

For smaller values of n (1000 or less), the running time is not significantly affected by the value of α . All running times are significantly less than one second (except for one outlier with $n = 1000, \alpha = 0.8$ with a running time of 1.27 seconds).

For $n = 10000$ (see Fig. 6.1), the median running time for $\alpha = 0.6$ is about twice as high as the time for the rest (0.47 seconds), whereas the median running time for $\alpha = 0.7$ and 0.75 is very similar (both around 0.25 seconds). The median for $\alpha = 0.8$ is 0.36 seconds, while the median for $\alpha = 0.9$ is 0.23 seconds. The running times of the graphs with $\alpha = 0.7$ are within a large range – the fastest runs are at 0.13 seconds, while the slowest are around 0.7 s. The range of running time is smaller for $\alpha = 0.6$ or 0.75, with between 0.28 and 0.84, and between 0.17 and 0.63, respectively. The running time for $\alpha = 0.8$ ranges between 0.11 and 1.03 seconds and for $\alpha = 0.9$ – between 0.08 and 0.38 seconds.

For $n = 100000$ (see Fig. 6.1), some trends start to become visible. The running time seems to decline, as the value of α increases. For $\alpha = 0.6$ all running times are between 0.7 and 6.84 seconds and the median running time is 1.92 seconds. A notable decline can be observed for $\alpha = 0.7$, where the running times are between 1.33 and 3.56 seconds. The median running time is 2.58 seconds. For $\alpha = 0.75$, the running time shrinks once again

with values between 1.32 and 2.59 seconds and a median of 1.55 seconds. For even larger values of α the running times drop even more, being between 1.03 and 2.27 seconds with a median of 1.88 seconds for $\alpha = 0.8$ and between 0.7 and 1.96 with a median of 1.41 for $\alpha = 0.9$.

As n grows, the gaps between the running times of graphs that have different alpha vales widen even more. Figure 6.1 includes the results for $n = 10^6$. The running times are between 25.69 and 53.48 seconds for $\alpha = 0.6$, 17.27 and 23.91 seconds for $\alpha = 0.7$, 12.84 and 18.3 seconds for $\alpha = 0.75$, 12.06 and 17.88 seconds for $\alpha = 0.8$, and 10.88 and 14.09 seconds for $\alpha = 0.9$. The median running time for $\alpha = 0.6$ is, predictably, the highest at 49.91 seconds. The medians for $\alpha = 0.7$, 0.75, 0.8, and 0.9 are, respectively, 19.66 and 17.25, 13.02, and 12.92 seconds.

The gaps grow even further for $n = 10^7$ – the median for $\alpha = 0.6$ is 600.8 seconds, 2.78 times more than the value for $\alpha = 0.7$ (216.2 seconds) and 3.04 times more than the one for $\alpha = 0.75$ (197.9 seconds). The medians for $\alpha = 0.8$ and 0.9 are, respectively, 156.1 and 149.4 seconds. All running times for $\alpha = 0.6$ are between 290.4 and 656.7 seconds, for $\alpha = 0.7$ – between 198.3 and 235.9 seconds, for $\alpha = 0.75$ – between 180.2 and 208.3 seconds, for $\alpha = 0.8$ – between 150.4 and 182.1 seconds, and for $\alpha = 0.9$ – between 134.9 and 164.3 seconds.

Figures 6.3 and 6.4 confirm the hypothesis that the running time of the algorithm is in $O(n \cdot \log^2(n))$. The figures show the running times, divided by $n \cdot \log^2(n)$ and the results do not appear to be significantly impacted by the value of n . However, the value of the divided running time declines for larger values of n , which implies that the theoretical bound is not tight and the algorithm is actually faster.

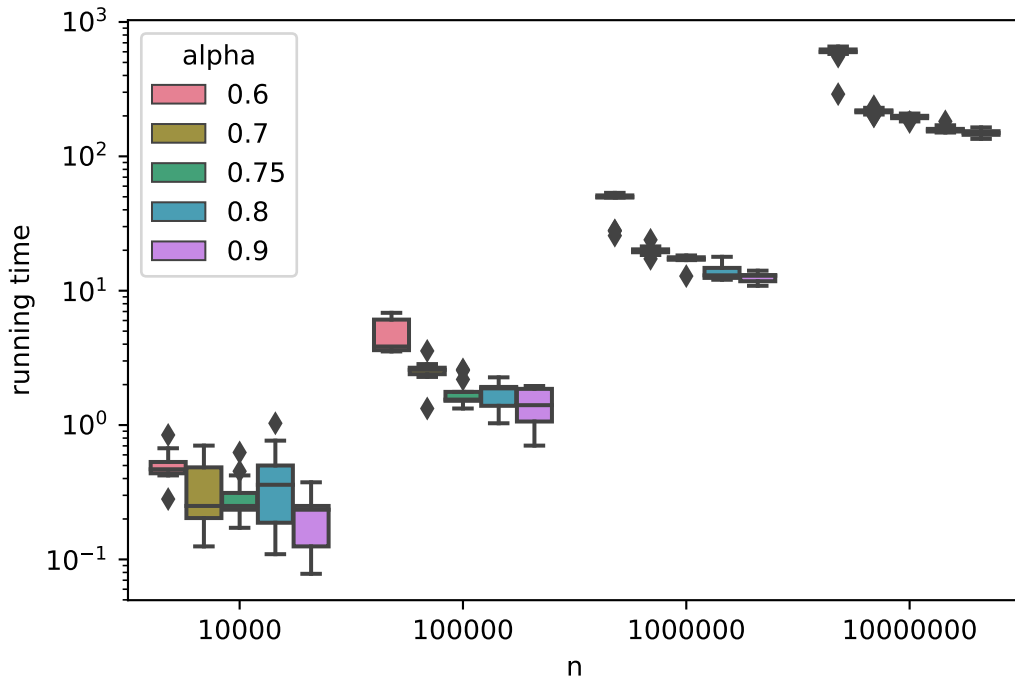


Figure 6.1: Group box plots for the running times of cases with various values for α for a growing n .

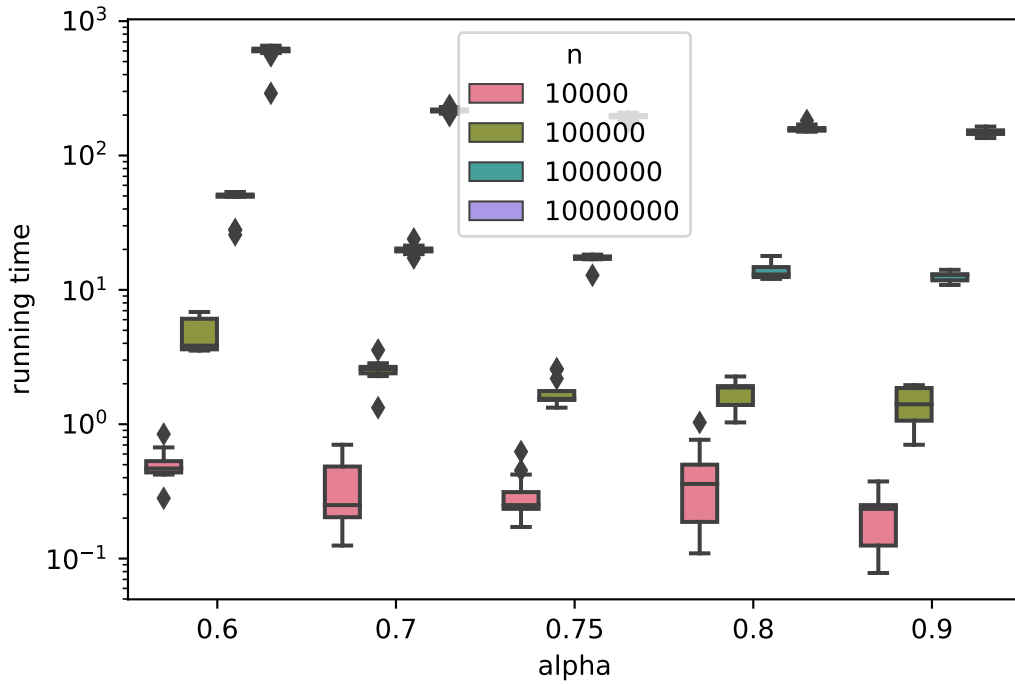


Figure 6.2: Group box plots for the running times of cases with various values for n for a growing α .

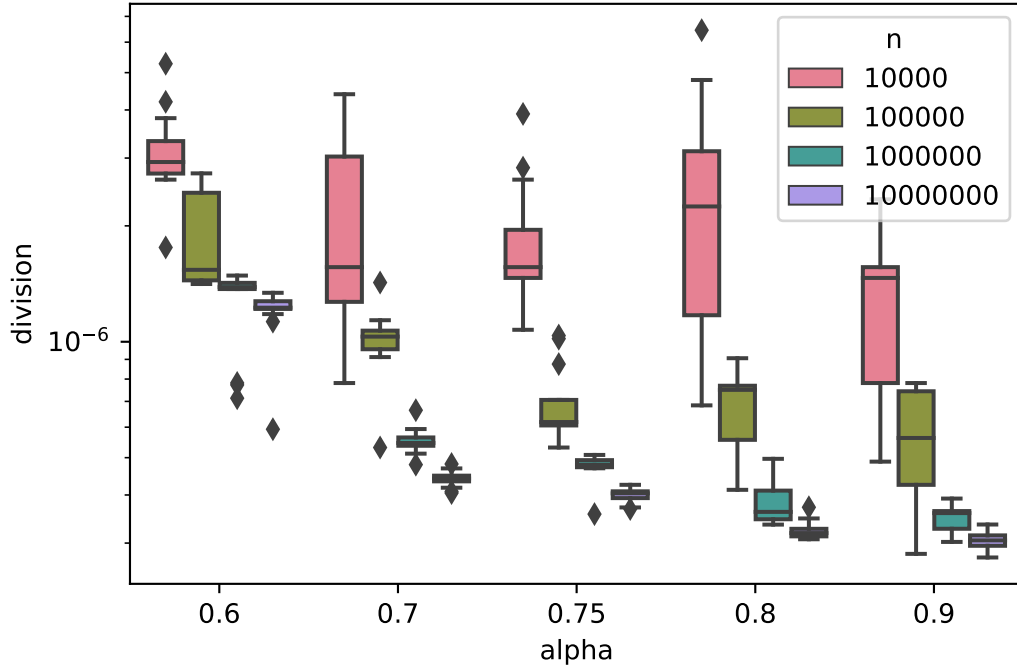


Figure 6.3: Group box plots for the running times of cases with various values for α for a growing n , where the running times are divided by $n \cdot \log^2(n)$.

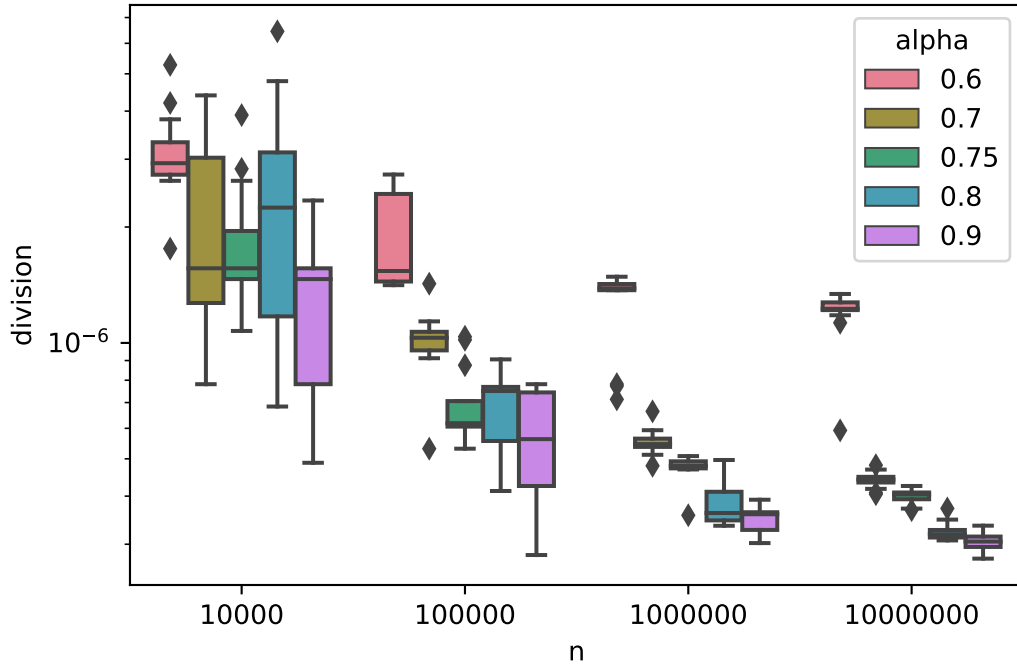


Figure 6.4: Group box plots for the running times of cases with various values for n for a growing α , where the running times are divided by $n \cdot \log^2(n)$.

6.2 Accuracy

In order to evaluate the accuracy of the hyperbolic distance function from `hypergirgs`, we need to calculate a more accurate approximation of hyperbolic distances and to compare it with the results from `hypergirgs`. We generated 1000 points at random. The coordinates of the points are polar. The radius of the each point is at most 16.0, whereas the angle is between 0 and 2π .

The distance between each point and all other points was calculated in two different ways. One of the computations, d_B , was done with the simple formula for calculating hyperbolic distance (Equation 3.2) where all mathematical operations were conducted with the help of functions from the `Boost` library, which provides higher accuracy. The other way that the distances were computed was by using the function d_H , available in the `Point.h` library of `hypergirgs` [BFK⁺19]. Whether the hyperbolic distance is computed via the function from `Point.h` or in an independently defined function, does not make a difference on the results.

Figure 6.5 shows the absolute error of the hyperbolic distance function from `hypergirgs`, as compared with the more highly accurate result, computed with the help of the `Boost` library. All absolute errors are less than 1×10^{-14} . No absolute error in the experiment was smaller than 1×10^{-21} . Almost a quarter, 22.3 %, of the errors are between 1×10^{-14} and 1×10^{-15} , 64.5% are between 1×10^{-15} and 1×10^{-16} , while only 11.8% are between 1×10^{-16} and 1×10^{-17} . Only 1.4% of the errors are smaller than 1×10^{-17} . Thus, the vast majority (98.6 %) of the errors is between 1×10^{-14} and 1×10^{-17} . That means that the absolute computational error that results from the distance function in `Point.h` is very small.

While data about absolute error sets a margin within which we can expect a result to differ from reality, often the magnitude of the error is dependent on the size of the input. For that reason, we also measured the relative error (how big the error is in comparison with the actual distance). The relative errors (see figure 6.6) are between 1×10^{-15} and 1×10^{-23} . Only few, 7.3 % of all errors, are between 1×10^{-15} and 1×10^{-16} , 80.5% are between 1×10^{-16} and 1×10^{-17} , 10.9% - between 1×10^{-17} and 1×10^{-18} . The rest of the relative errors make up 1.3 %. Again, most of the errors are in one of three blocks on the histogram – 98.7 % of the errors are within 1×10^{-15} and 1×10^{-18} (each block represents one exponential step). Although the relative error appears most concentrated between 1×10^{-16} and 1×10^{-17} , 86.8 % of the absolute errors are less than 1×10^{-16} , as we can see from figure 6.5. That is evidence that the bigger absolute errors are likely due to larger inputs.

6.3 Filtering

We also tested how many of the edges that end up in the random HMST are generated in each iteration. Our tests show that the vast majority of the edges that are collected in the result are generated in the first iteration. However, the total number of edges generated does not decline with every iteration. In fact, it even becomes gradually higher. We computed the median and the average data over 30 random graphs. Figures 6.7 and 6.8 illustrate the results. The number of iterations necessary to generate the random HMST drops for larger values of α . Most samples for $\alpha = 0.6, 0.7$ or 0.75 require 4 or 5 iterations (although very few edges from the fifth iteration stay on the MST), whereas the samples for $\alpha = 0.8$ or 0.9 require only 3 iterations.

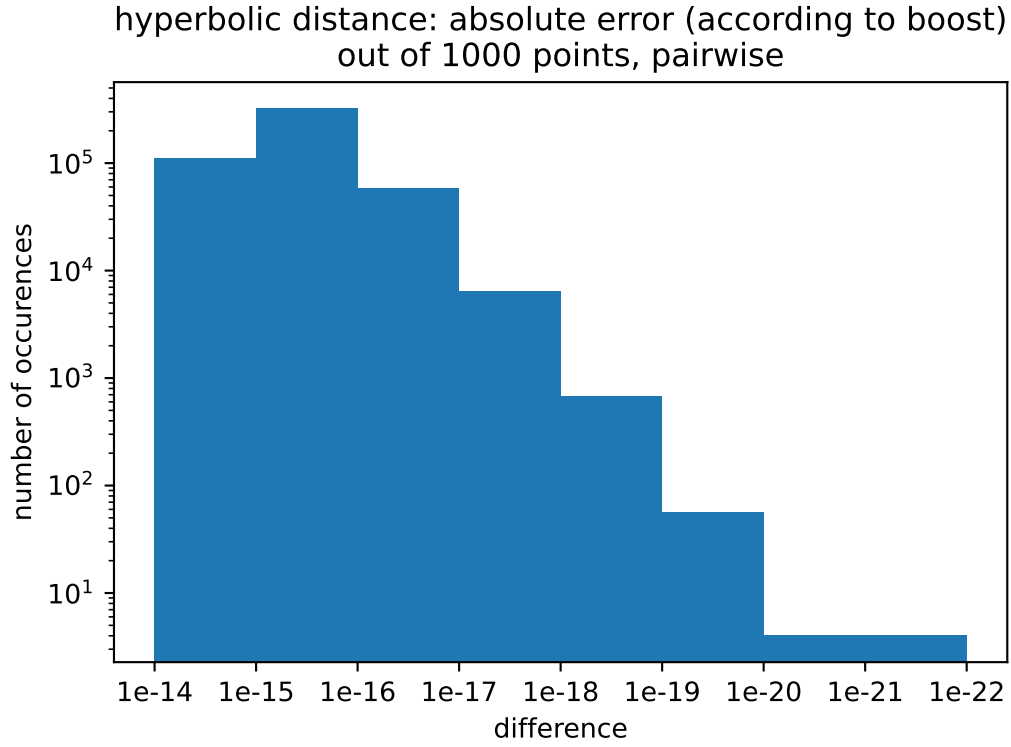


Figure 6.5: This figure shows the absolute difference between the results computed with the `hypergirgs` function and the ones computed with Boost.

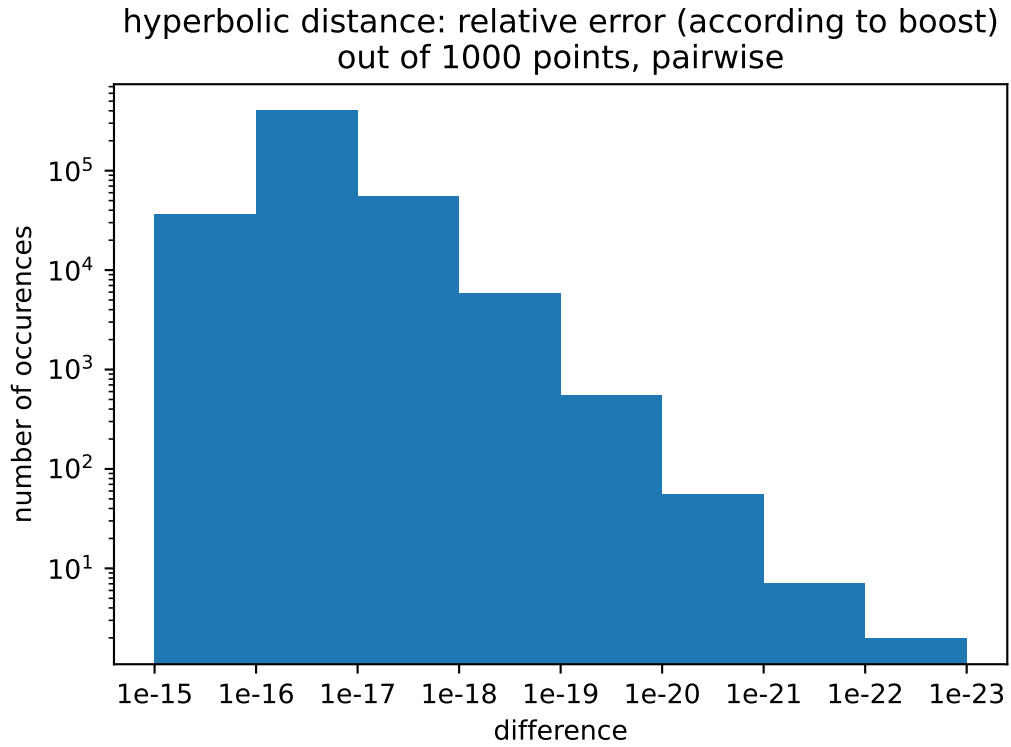


Figure 6.6: This figure shows the absolute difference between the results computed with the `hypergirgs` function and the ones computed with Boost, divided by the results computed with Boost.

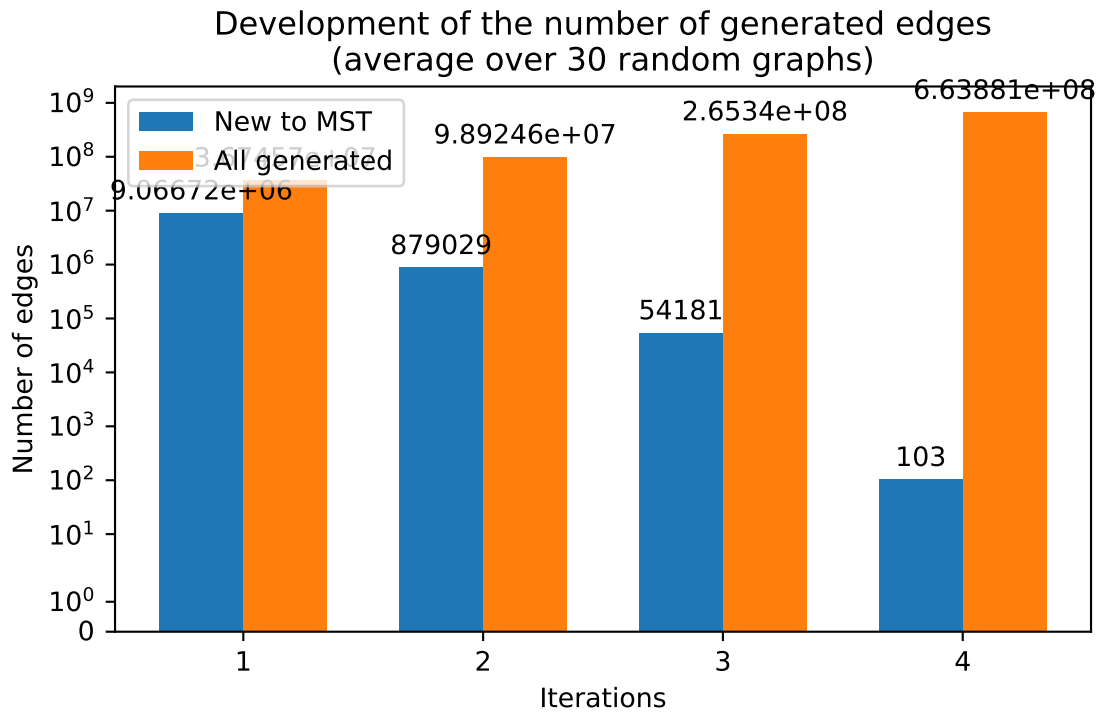


Figure 6.7: This figure shows the average number of edges generated against the average number of edges added to the MST on each iteration of the algorithm.

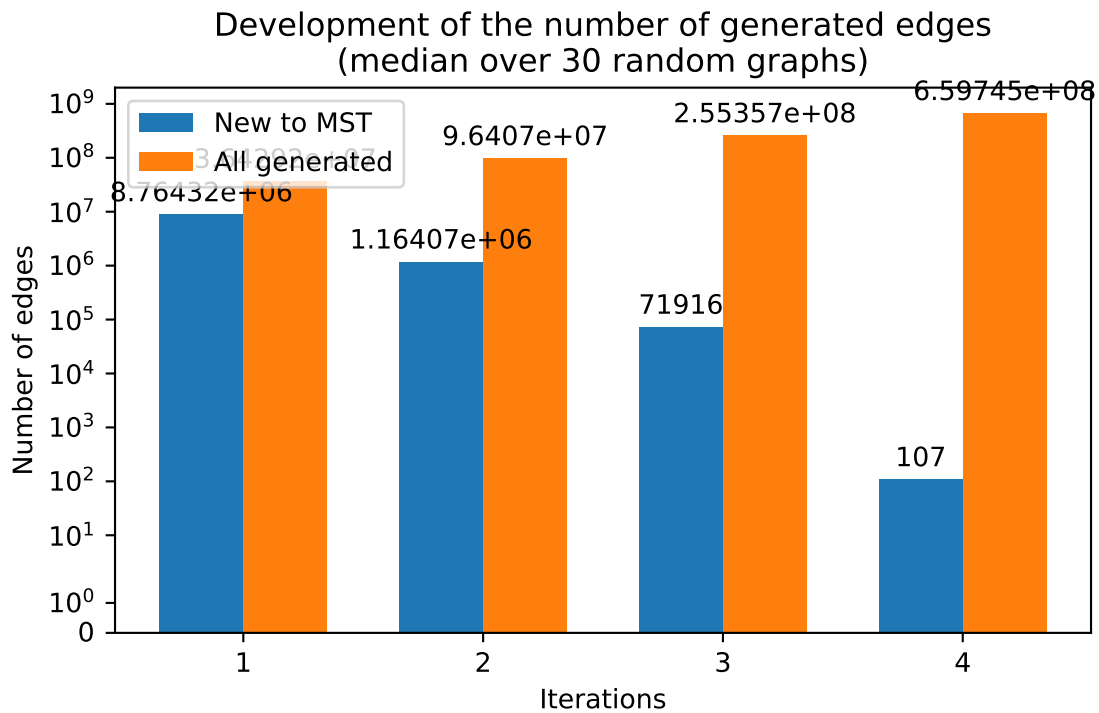


Figure 6.8: This figure shows the median number of edges generated against the median number of edges added to the MST on each iteration of the algorithm.

6.4 Degree Distribution

Finding how similar HRGs are to hyperbolic MSTs is important. If they do share similarities, then it is possible that some of the rules for HRGs apply for hyperbolic MSTs as well. To do that, a good first step is to identify whether the degree distribution of hyperbolic MSTs follows a power-law. In this chapter, we examine a case study of one random hyperbolic minimum spanning tree. The number of nodes on which we generated it is 10^7 .

The node with the highest degree – the biggest “hub” of the minimum spanning tree has 139855 neighbors. Therefore, there is a node, connected to almost 1.4% of the other nodes. The next 5 highest degrees are considerably lower: 52637, 39732, 39226, 35674, 33264. After that, the differences start to become less significant. On the other end of the spectrum, as evident in Fig. 6.9, 75.43% of the nodes are leaves, 14.34% have only two neighbors, 4.39% have three, and 1.92% have four. That means that the nodes which have five or more connections make up slightly less than 3.93% of the entire set of nodes.

Fig. 6.10 displays the correlation between degree and the fraction of the nodes that have that degree and that correlation is inverse. Our initial goal was to generate heterogeneous trees, due to the heterogeneity exhibited in transmission trees. The results show that we achieved this goal successfully. As the line on the graph is fairly straight, it appears that the degree density follows a degree distribution similar to a power-law function, resembling one exhibited by a HRG. That suggests that the MSTs which result from the algorithm are heterogeneous.

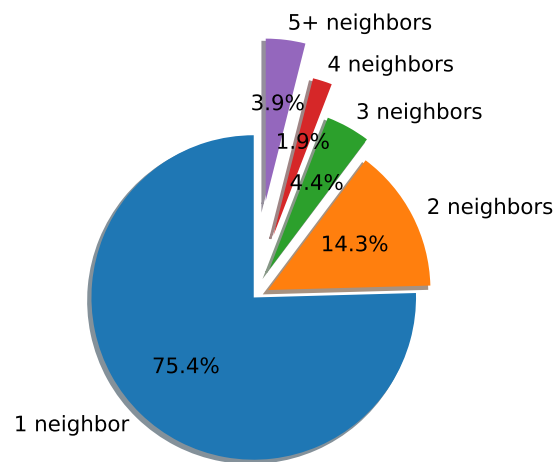


Figure 6.9: A pie chart showing the percentage of nodes with 1, 2, 3, 4 and 5 or more neighbors.

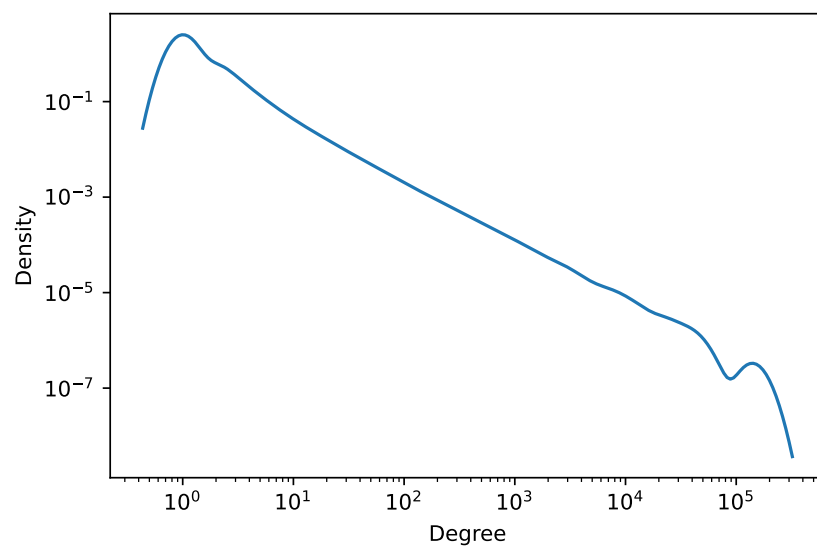


Figure 6.10: A line plot that shows the degree distribution

7. Conclusion

We have defined, implemented and tested a new algorithm for generating random hyperbolic MSTs that works in $\Theta(n \cdot \log^2(n))$. Furthermore, the graph can be extracted as a graphic. The degree distribution of a sample random HMST closely resembles a power-law distribution. Moreover, the hyperbolic distance function in `hypergirgs` shows high accuracy with absolute errors never exceeding 1×10^{-14} , compared to the same distances, computed with the `Boost` library.

HRGs can help us understand and predict the world around us and, especially considering future pandemics, it would be interesting whether our random HMST generation algorithm and its implementation can be beneficial in helping professionals understand transmission trees better.

As the code for the visualization of the hyperbolic MST is not dependent on our algorithm for generating the MST that is being visualized, it can be easily reused if researchers develop another algorithm for generating hyperbolic MSTs and want to visualize them on the Euclidean plane.

While the running time of the algorithm is not optimal, we expect it to be more numerically stable than existing algorithms, which is typically more important. However, this still should be investigated. Due to MST being a classical problem in algorithmics, this can open the door for more numerically stable implementations of other problems on the hyperbolic plane.

Bibliography

- [AESW91] Pankaj K. Agarwal, Herbert Edelsbrunner, Otfried Schwarzkopf, and Emo Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete Comput. Geom.*, 6(1):407–422, dec 1991.
- [BFK⁺19] Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. Efficiently generating geometric inhomogeneous and hyperbolic random graphs, 2019.
- [BP16] A.L. Barabási and M.Á. Pósfai. *Network Science*. Cambridge University Press, 2016.
- [CCK10] Samidh Chatterjee, Michael Connor, and Piyush Kumar. Geometric minimum spanning trees with geofilterkruskal. In Paola Festa, editor, *Experimental Algorithms*, pages 486–500, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [ER59] P. Erdős and A. Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290, 1959.
- [GPP12] Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. Random hyperbolic graphs: Degree sequence and clustering, 2012.
- [HP08] George T. Heineman and Gary Pollice. *Algorithms in a Nutshell*. Shroff Publishers & Distributors Pvt Ltd, December 2008.
- [JSC⁺21] Zhou J, Dong S, Ma C, Wu Y, and Qiu X. Epidemic spread simulation in an area with a high-density crowd using a seir-based model. *PLoS One*, e0253220(16(6)), 2021.
- [KPK⁺10] Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Physical Review E*, 82(3), sep 2010.
- [Kru56] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In *Proceedings of the American Mathematical Society*, 7, 1956.
- [NMN01] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Math.*, 233(1–3):3–36, apr 2001.
- [Pen03] M. Penrose. *Random Geometric Graphs*. Oxford studies in probability. Oxford University Press, 2003.
- [Pri57] R. C. Prim. Shortest connection networks and some generalizations. *The Bell Systems Technical Journal*, 36(6):1389–1401, 1957.
- [VS16] Kevin Verbeek and Subhash Suri. Metric embedding, hyperbolic space, and social networks. *Computational Geometry*, 59:1–12, 2016.

- [WDE⁺97] M.E.J. Woolhouse, Christopher Dye, Jean François Etard, T A Smith, Jacques D Charlwood, Geoffrey P. Garnett, Patrick Hagan, Jeffrey L. K. Hii, P D Ndhlovu, Rupert J Quinell, C. H. Watts, S Chandiwana, and Roy M Anderson. Heterogeneities in the transmission of infectious agents: implications for the design of control programs. *Proceedings of the National Academy of Sciences of the United States of America*, 94 1:338–42, 1997.
- [YBS⁺12] R. J. F. Ypma, A. M. A. Bataille, A. Stegeman, G. Koch, J. Wallinga, and W. M. van Ballegooijen. Unravelling transmission trees of infectious diseases by combining genetic and epidemiological data. *Proc Biol Sci*, 279(1728):444–450, 2012.