2024

# VISUALIZATION OF HIGH DIMENSIONAL DATA IN LOW DIMENSIONAL SPACE VIA CLUSTERED MANIFOLD MAPPING

David Perrone
*University of Rhode Island*, david_perrone@uri.edu

# VISUALIZATION OF HIGH DIMENSIONAL DATA IN LOW DIMENSIONAL SPACE VIA CLUSTERED MANIFOLD MAPPING

BY

DAVID PERRONE

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2024

MASTER OF SCIENCE

OF

DAVID PERRONE

APPROVED:

Thesis Committee:

Major Professor Noah Daniels

Shaun Wallace

Gretchen Macht

Jean-Yves Hervé

Brenton DeBoef
DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
2024

## ABSTRACT

Visualizing high dimensional data can be a challenging task due to the difficulty people face in comprehending information beyond three dimensions. Further research and development of tools in this area could prove valuable for creating efficient, intuitive, and accurate visualizations. It could also provide insight into the manifold hypothesis, which suggests that high dimensional data can exist in low dimensional space.

This thesis proposes the utilization of clustered manifold mapping as a novel visualization technique that summarizes a dataset into a hierarchal tree of clusters by partitioning the data based on a user-specified distance metric. A subset of clusters can be carefully selected from the tree to create a 3D graph using the Unity game engine, which enables the user to interact with the and explore various features of the data.

The graphs produced with this approach will be quantitatively and qualitatively compared with existing methods such as UMAP, which demonstrates its contributions to the field of visualizing high dimensional data. Furthermore, visualizing the tree of clusters in addition to the graph provides a greater understanding into the field of clustered manifold mapping.

## ACKNOWLEDGMENTS

First and foremost, I extend my heartfelt gratitude to Dr. Noah Daniels for his unwavering assistance and dedication this past year. Without his invaluable guidance, this thesis would not have been possible. I am sincerely grateful for his mentorship and support.

Noah's ABD research group offered invaluable support, advice, and encouragement throughout my project, particularly during the demonstration phases of my code base. Their guidance was instrumental in refining my work and navigating challenges effectively.

Among those research members I would especially like to thank Najib Ishaq, whose mentorship and knowledge of CLAM and Rust were invaluable. I am grateful for his patience and guidance whenever I encountered challenges.

I would like to acknowledge Andrew Lefebvre and Joey Buono for their valuable contributions to my codebase. Joey's implementation of a force-directed graph in Rust provided a crucial foundation for my own work, while Andrew's assistance in refactoring the CHAODA and CLAM codebases was immensely helpful. I am also grateful to the ABD research group and their commitment to maintaining the CLAM codebase.

I extend my appreciation to Shaun Wallace for his insightful feedback on my UI design and for encouraging me to conduct HCI experiments.

Finally, I would like to express my gratitude to my friends and family for their unwavering support and encouragement throughout this journey. Their

faith in me, even when the subject matter seemed obscure, was a constant source of motivation.

## TABLE OF CONTENTS

## LIST OF TABLES

TABLE PAGE

## LIST OF FIGURES

# CHAPTER 1

## INTRODUCTION

Visualizing high-dimensional data can be challenging due to the curse of dimensionality, which refers to issues such as computational complexity and the challenge of representing data beyond three dimensions in a way that humans can easily understand [1]. Modern datasets are growing increasingly larger, both in the number of data points and the number of features describing them. For example, if a dataset is described by two features, one could create a 2d line or bar plot and display the information in a meaningful way. However, if the data is described by hundreds or thousands of features, it is difficult to get an intuitive feel for what the data looks like [2].

A common method for visualizing such high dimensional datasets is to simplify the data by finding a lower dimensional structure in it [3]. To achieve this, manifold learning algorithms are commonly employed to calculate measures of the local geometry of the manifold, after which the original data points are no longer needed [3]. A prevalent visual representation of the manifold, exemplified by UMAP, TMAP, and IsoMap, is in the form of a weighted graph [4]. This tool will build on these existing visualization methods that have various tradeoffs in performance, quality of the manifold, and user interface features.

## 1.1 Problem Statement

The weighted graphs used in common visualization methods such as UMAP and t-SNE are created using a nearest neighbor approximation. The nearest neighbor algorithm is used to approximate the local structure of the manifold by showing a cluster in relation to other clusters that it shares some features with. The relationships between disconnected neighborhoods in the graph represent the global structure in the manifold. These algorithms must perform a balancing act of trying to preserve both the local and global structure while visualizing the manifold.

This low dimensional representation of the manifold is estimated by these algorithms using a force directed graph. Based on some “distance” calculated between data points in high dimensional space, the goal is to position clusters in 2D or 3D space such that the distance between them is representative of their relationship in the original embedding space. This is done by applying attractive or repulsive forces between clusters until their positions have settled in an optimal layout. The balancing act mentioned earlier comes from applying forces between the disconnected portions of the graph without disrupting the local relationships within the neighborhood.

## 1.2 Proposed Solution

The notable divergence of this work and existing methods is the use of CLAM (Clustered Learning of Approximate Manifolds) to construct the graph. CLAM uses a clustering process that repeatedly divides the dataset and places datapoints that are estimated to be related into the same cluster. The

graph is then constructed by selecting a subset of clusters from the tree such that the entire dataset is represented. This accomplishes one of the first challenges of attempting to distill a dataset into a subset of its features.

The selection of clusters to represent the dataset in the graph is a challenge that can be solved using CHAODA, a collection of anomaly detection algorithms created by training meta-machine learning models according to several geometric and topological properties [5].

CLAM also provides a diverse set of distance functions that can be used to estimate the distance not only between datapoints but between clusters. In addition, it provides a radius for each cluster, which describes the greatest distance from its geometric median to any datapoint in the cluster. By leveraging the distance functions and radii of clusters, the edges in the graph can be created by looking for an “overlap” between clusters.

In addition to the exploration of the manifold through clustered manifold mapping, this work looks to create an immersive and interactive experience for the user. The visualization will be created using the Unity game engine, which will allow for efficient rendering of the tree and graph. In addition, the ability for the user to move around the graph with a camera and interact with clusters will provide insights into the underlying manifold of the dataset.

### 1.3 Findings

The results from this work look promising as the graphs produced for datasets can be tuned based on a variety of parameters to view low and high resolutions of the dataset. Selecting clusters from various depths in the tree

allows the user to manage the balance of the local and global structure of the graph. In addition, the visualization of the tree created by CLAM provides an insight into the density of various sections of the manifold.

The accuracy of the low dimensional representation is quantified using a series of tests that measure distortion between the geometry of the low dimensional manifold and the representation of the data in high dimensional space. Initial results show that the accuracy of the graph improves with each iteration of the force directed algorithm used to create the 3D layout and that the graphs produced provide a better representation of the distances between points than in UMAP.

# CHAPTER 2

## REVIEW OF LITERATURE

### 2.1 The Curse of Dimensionality

Datasets over the past few decades have been growing not only in the number of instances of data but in the number of features describing them [6]. An example of an industry with growing datasets is biotech; which needs to advance its capability to analyze, visualize, and interpret data in order to better understand diseases [7]. Another cause of the increase in dimensionality can be seen in the analysis of images and even entire movies, where a single observation could have dimensions in the thousands or billions [6].

Donaho provides an example in his article asking the reader to consider a cartesian grid on a unit cube in 10 dimensions with grid spacing of 1/10 that contains $10^{10}$ points. An exhaustive search of this space could result in attempting billions of evaluations. In his lecture titled “The Curses and Blessings of Dimensionality”, Donaho uses the curse of dimensionality “to refer to the apparent intractability of systematically searching through a high-dimensional space, the apparent intractability of accurately approximating a general high-dimensional function, the apparent intractability of integrating a high-dimensional function [6].” He then confidently states that high dimensional data analysis will be a significant activity leading to the development of new methods in the coming years.

### 2.2 The Manifold Hypothesis

In contrast to the Curse of Dimensionality are the blessings of dimensionality. In Gohan's article on the Blessings of dimensionality, they state that in the world of statistical mechanics, a complex system can be presented as a union of many weakly interacting subsystems that exist in lower dimension. Furthermore, contributions from physicists and mathematicians have shown that random points in a high dimensional sphere tend to lie near the surface [8]. While these theorems relate to thermodynamics and behaviors of particles in gas, mathematicians theorized that the behavior of particles in high dimensional balls could be related to the behavior of high dimensional data. This leads to a collection of methodologies for analyzing high dimensional data based on the hypothesis that real-world data tend to lie near a low dimensional manifold, called manifold learning. The underlying hypothesis is referred to as the manifold hypothesis [9].

## 2.3 Manifold Learning

Manifold learning is a recent approach to nonlinear dimensionality reduction based on the idea that data points described by thousands of features may be described as a function of a few key underlying parameters [2]. A common example of such data is to consider several images taken of an object simultaneously from various angles. While the images may contain hundreds of dimensions, they would also contain a significant amount of overlap in their data. If one had to analyze hundreds of these images, it would be helpful to get a simplified representation such that the images are

described by key underlying features that describe their similarities or differences.

This idea can be formalized using the manifold hypothesis, which assumes that data lie along a low-dimensional manifold embedded in high-dimensional space. In other words, we do not need to keep all the features of the data to compare the images. Attempting to uncover this manifold structure in a data set is referred to as manifold learning [2].

### 2.3.1 Manifold Learning Algorithms

Here, I will provide a summary of the common steps found in manifold learning algorithms that lay a foundation for existing methods section, where I will go into more detail on common manifold learning techniques. One of the first manifold learning algorithms, which I will use as an example here, is Isomap. Its algorithm consists of two main steps. The first is to estimate the distance between points in the input data. The next step is to find points in a lower-dimensional Euclidian space such that the distance between the points match their distance in the original embedding space [2]. Manifold learning algorithms are then forced into a balancing act where they need to consider the local and global structure of the manifold.

Isomap is considered a global method because it constructs an embedding from the distance between all pairs of points. A local method of manifold learning would only consider the distance between a point and its immediate neighbors. This tradeoff leads to distinguishing characteristics of common manifold learning techniques. For example, assuming that each point

in the dataset is assigned some number of “neighbors” that are assumed to be related in some way, a local method would do a good job of representing the distance between points in a local neighborhood. However, non-neighboring points could be found in locations in the Euclidian space that are much closer to each other than they are in the embedding space. Conversely, a global method would tend to do a good job of spacing non-neighboring points into distinct clusters but would fail to accurately represent the relationships of datapoints in the neighborhood [2].

Another factor to consider is that most manifold learning techniques take in the number of neighbors as a parameter, which can vastly affect the accuracy of the dimensionality reduction. In addition, it can be difficult to prove that the manifold being represented actually exists and then even quantifying how accurate of a representation your manifold is to the real manifold is a challenge [2].

## 2.4 Force Directed Graphs

Manifold learning algorithms and visualizations typically use a graph to represent the underlying data. The specifics might differ between certain visualization methods, but the overall concept is that a graph is formed where vertices in the graph represent data points and edges in the graph represent relationships between vertices and their neighbors. Here, I will cover the overall concept of how force directed graphs work before going into how they are used specifically in visualizations.

Force directed algorithms in general are a common way to visualize graphs because certain heuristics and hyper parameters can be used to achieve desired stability and readability properties [10]. These algorithms can be broken down into two phases. The first is an initialization phase, where certain constant values are set as well as an initial layout of the nodes is formed. Many algorithms such as the Fruchterman and Reingold algorithm will randomly initialize the node positions [10]. The next step is the iterations where forces are applied to edges and/or nodes in the graph repeatedly until some termination condition is met. Ideally, the graph layout should converge towards some desired end state.

In Fruchterman and Reingold's implementation, repulsive forces are calculated between every pair of vertices, but attractive forces are calculated only between vertices that share an edge. They made this decision because they decided to emphasize the importance of the local layout of the graph. By only applying attracted forces along edges, they encourage a vertex to only be located nearby other neighbor vertices. This means that their algorithm would lead to what is considered a local embedding in manifold learning. Other algorithms exist that try to find an ideal distance between vertices and non-neighbors which would lead to a balancing act between the global and local structure of the graph. One such algorithm was developed by Kamada and Kawai, who defined the ideal distance between disconnected vertices as being proportional the length of the shortest path between them [11]. The implication of a path existing between them is especially important in the context of my

work because this would still only affect the structure within disjoint graph components.

“A disjoint graph component is a connected subgraph C of the graph G which is not properly contained in any other subgraph of G [12].” Another expression of this could be that if there are two components in a graph and a vertex *v* exists in the set of vertices of one component but not in the set of the other component, they are said to be disjoint [12]. This is notable because the ideal layout discussed above only applies attractive forces within the components and does not consider one disjoint component relative to another. As such, a layout would ideally have a better representation of the local geometry of the manifold but would still not accurately represent the global structure as there would only be repulsive forces between disjoint components.

The goal of Reingold and Fruchterman’s force directed algorithm was twofold: “Vertices connected by an edge should be drawn near each other and vertices should not be drawn too close to each other [11].” However, they do note that some graphs could be too complex to draw attractively. A notable aspect of many force directed graph algorithms such as Fruchterman and Reingold’s is that they have a target frame that should contain the graph. As such, they create four imaginary walls representing the frame that will prevent any vertices from leaving the frame. This could lead to a problem if a manifold learning force directed algorithm wants to accurately represent relationships between data points but needs to confine the data within a certain frame and simultaneously prevent vertices from being too close to each other. This topic

will be covered in more depth in the review of t-SNE as they discuss the tradeoffs between aesthetics and accurately portraying the geometry of the manifold.

As mentioned earlier, a common step in force directed algorithms is to iterate some number of times and apply forces to the graph until some stop criteria is met. This stopping criterion is not so easily defined and varies from algorithm to algorithm. For example, “Eades simply asserted that ‘almost all graphs reach a minimal energy state after the simulation step is run 200 times’ [11].” Some algorithms such as Kamada and Kawai have a target state that they aimed for their graph to achieve but did not explicitly state the number of iterations to achieve that state as it could vary depending on the dataset. Frechterman and Reingold stated in their paper that the number of iterations to be used in their algorithm is “guesswork” [11].

## 2.5 Existing Visualization Methods

### 2.5.1 PCA

Principal Component Analysis is a dimensional reduction technique first discussed by Pearson in 1901 but it took decades before the available computing power made it feasible to use on datasets [13]. Its primary goal is to reduce the dimensionality of a dataset while retaining as much “variance” in the dataset as possible. By doing so, it can be used to identify patterns in data that highlight the similarities and differences between various datapoints [14].

Principal Component analysis does this by computing a covariance matrix that represents the correlation between each combination of datapoints.

If their covariance is positive, they are positively correlated and if their covariance is negative, they have an inverse correlation (as one value increases, the other decreases). The covariance matrix is then used to compute eigenvectors and eigenvalues where the former represent the directional vectors on which data lie and the latter represent the importance of the vector in representing the dataset. Principal Components can then be created as linear combinations of the original dataset weighted by their corresponding eigenvectors. The least important principal components can then be discarded to summarize the dataset while reducing its dimensionality [13].

This leads to a simplified description of the dataset that can be used to analyze the structure in a lower dimension. It can also be thought of as an unsupervised learning method that finds patterns in datasets without references to prior knowledge of the grouping of datapoints [14].

PCA is a powerful tool that comes with limitations such as the fact that the underlying structure of the data must be linear and that patterns that are highly correlated could be unresolved because PCA tries to create uncorrelated components from the data [14].

### 2.5.2 t-SNE

T-SNE is a manifold learning technique first introduced in 2008 that aims to visualize data by giving each datapoint a location in 2d or 3d space. It does this by converting high dimensional Euclidean distances between datapoints into conditional probabilities that represent similarities. These

similarities are then used for each datapoint to find other datapoints that could be considered its neighbor. T-SNE then initializes a force directed graph in 2d or 3d space by providing each datapoint a random location.

That same probability of datapoints being neighbors can then be used to estimate the “distance” between the high dimensional and low dimensional representation of the dataset. For simplicity, let’s assume that the number of nearest neighbors being found is a small number such as three. We will then assume for each datapoint in high dimensional space, some datapoints u,v,w were found to be the nearest neighbors. For each datapoint in lower dimensional space, we would also search for its three nearest neighbors. If u’, v’, w’ correspond to the same datapoints, then the difference between the datasets would be minimal. In more technical terms, t-SNE “minimizes the sum of the differences over all datapoints using a gradient descent method, whose cost function focuses on retaining the local structure of the data in the map [15].” To minimize this distance, attractive or repulsive forces are applied between datapoints in lower dimensional space based on if their distance between datapoints is greater or less than the similarity of the datapoints in high dimensional space.

T-SNE also scales the forces applied along the “springs” of its graph so that longer springs will apply more force than shorter springs. This is accomplished by exerting force proportional to the difference in similarity between the points in each dimensional space. If the difference is 0 for example, it means the datapoints in low dimensional space perfectly represent

the relationship between the datapoints in high dimensional space, and no forces would be applied between them [15].

In their paper, Laurens van der Maaten observe that in high dimensions, there are numerous ways that datapoints can have the same distance. However, as you reduce the number of dimensions, you reduce the number of ways that distance can be represented. This means that if all datapoints were equidistant to each other, the produced 2D graph could be overcrowded. They note that if small distances are to be represented accurately, then moderate or large distances could be placed too far away to fit properly in the 2d dimensional map.

To mitigate this issue, they apply a slight attractive force between a datapoint and the far away datapoints with the intention of drawing clusters closer together to fit on the map. An unfortunate side effect of this is that these slight forces can add up and cause too many clusters to be pulled towards the center of the map, resulting in overcrowding. To account for this, they follow a method presented by Cook et al. in 2007 that creates slight repulsive forces that prevent the distance between two datapoint falling below a certain threshold. This essentially creates a minimum possible distance between datapoints in the 2d map that fight overcrowding while the attractive forces still prevent clusters from disappearing from the map [15].

### 2.5.3 UMAP

UMAP is another manifold learning technique for visualization high dimensional data that is constructed from a theoretical framework based in

Riemannian geometry and algebraic topology. In their paper published in 2020, the authors argue that it has better run time performance and preserves more of the global structure of the manifold than t-SNE. They also state that neighborhood based manifold learning algorithms should select their fundamental components through well-grounded theoretical decisions [4].

The UMAP paper has a chapter in it that goes into detail on the mathematical theory behind their algorithm. For the purposes of this review, I will skip over the math theory and focus more on the higher-level details. At a high level, the UMAP algorithm looks quite like t-SNE. They create a topological representation of their dataset in high dimensional space and then construct a topological representation in a lower dimensional space. UMAP then uses a force directed algorithm to minimize the cross-entropy between the two topological representations [4].

Like t-SNE, UMAP uses a k-nearest neighbor algorithm to represent the local structures of the manifold. It then estimates the geodesic distance between each point and its k neighbors to represent the edges of its graph. In summary, UMAP can be described “in terms of, constructions of, and operations on, weighted graphs [4].”

The first step of UMAP’s algorithm will sound familiar: to construct a weighted k-neighbor graph from the dataset. The second is to compute a low dimensional layout of said graph. The key difference between the t-SNE and UMAP is that the latter’s cost function used to minimize the difference between

the two topological representations is more efficient, resulting in performance gains.

The behavior of their force directed graph can be described as applying attractive forces along edges of the graph and repulsive forces between nodes. However, applying repulsive forces between all nodes as seen by Reingold and Fruchterman results in a time complexity of $O(E) + O(N^2)$ [11]. Therefore, the UMAP algorithm reduces the time complexity by randomly sampling repulsive forces from vertices whenever an attractive force is applied along an edge.

The UMAP algorithm takes in four hyper parameters: the number of neighbors to consider, the target embedding dimension, the minimum distance between datapoints, and the number of epochs. The number of neighbors considered has an impact on the tradeoff of local vs global manifold learning performance. The authors state that “smaller values will ensure detailed manifold structure is accurately captured (at a loss of the “big picture” view of the manifold), while larger values will capture large scale manifold structures, but at a loss of fine detail structure which will get averaged out in the local approximations [4].”

Another hyperparameter of note is the minimum distance, which deals with the issue of “overcrowding” as described in t-SNE by allowing the user to specify the how closely together points can be packed in the lower dimensional graph. The authors note that this is more of an aesthetic choice for helping readability of visualizations as larger values will force the

embedding to spread the points out, possibly leading to a loss in accuracy in the representation of the manifold.

The authors of the UMAP paper conclude their findings by pointing out limitations of their algorithm. One observation they make is that UMAP tends to focus on the local structure of the geometry of the manifold and that UMAP may not be the best choice if one's primary goal is to visualize the global structure. The authors of UMAP also note that UMAP is focused on preserving the topology of the structure rather than pure metric structures [4]."

In their future works section, the authors note that they attempt to discover a manifold on which the data is uniformly distributed. Thus, if the data consisted of a loose structure in one area and a densely packed structure in another area, UMAP would put these local areas "on even footing [4]." The authors also echo what Cayton described in his paper: that "there is a lack of clear objective measure, or even definitions, of global structure preservation [4]."

## 2.6 Clustered Manifold Mapping

### 2.6.1 CLAM

The notable difference between the manifold learning technique proposed in this thesis and the methods summarized in the previous section is that it uses clustered manifold mapping to create its graph rather than k-nearest neighbors. Clustering is a field that aims to arrange an unordered collection of objects such that nearby objects are similar [6]. This is usually achieved by grouping similar datapoints into the same cluster [5]. The

CHAODA paper notes that the term manifold learning is “largely synonymous with dimension reduction and proposed manifold mapping to refer to the study of the geometric and topological properties of manifolds in their original embedding spaces [5].” As such, the authors proposed Clustered Learning of Approximate Manifolds (CLAM) as a novel technique.

CLAM defines a cluster as a set of points with a center and radius. The center is the geometric median of the points grouped in the cluster while the radius is the greatest distance from the center to any point in the cluster [5]. CLAM uses these clusters to create a tree representation of the dataset where each non-leaf cluster has two child clusters.

CLAM creates this cluster tree using a “divisive hierarchical clustering algorithm [16].” The initial step in the partitioning of the tree is to take a cluster containing |C| points (the cardinality of the dataset) and randomly sample the square of |C| points. For each of these randomly selected points, the pairwise distance between all points are calculated based on some metric distance function. The geometric median of the cluster is then calculated by minimizing the sum of the distances to all other points in the sample. CLAM then designates a left “pole” of the cluster as the datapoint that is farthest from the geometric median as well as a right pole that is the cluster farthest from the left pole [16]. The cluster is then partitioned such that data points that are closest to the left pole are assigned to the left child and clusters closest to the right pole are assigned to the right child. If any clusters are equidistant from

the two poles they are assigned to the left child by default. This often leads to an unbalanced tree that leans to the left.

The authors note that this is a positive feature of the algorithm as the "varying sampling density in different regions of the manifold and low dimensional shape of the manifold itself will cause it to be unbalanced [16]." The authors only expect a perfectly balanced tree if the dataset is uniformly distributed. This contrasts with the UMAP algorithm which assumes that the data is uniformly distributed on the manifold.

### 2.6.2 Clusters

CLAM's clustering process also provides the advantage of memoizing properties of the clusters such as radius as defined above, cardinality (the number of data points stored within the cluster), local fractal dimension (defined as an approximation of the dimensionality of the lower-dimensional manifold in the "vicinity" of a given point", as well as an offset to access points in the dataset [16]. The data corresponding to a cluster can be found in the range *[offset, offset* + *cardinality]*. These properties can be leveraged in the visualization to provide more details about clusters and the underlying manifold.

### 2.6.3 Graphs

The cluster tree described above plays an important role in the visualization, however the most important contribution to the visualization is the ability to induce graphs that can be created by mapping specific clusters in

a tree to vertices of a graph. Edges in the graph are drawn between any two vertices whose corresponding clusters have overlapping volumes, “i.e. the distance between their centers is less than or equal to the sum of their radii [5].”

Clusters can be selected from the graph based on several properties such as their depth in the tree, cardinality, radius, etc. Clusters at lower depths in the tree can be considered at a “lower resolution” than those at greater depths. The authors note that inducing a graph across a variety of depths, as seen in figure 1, “efficiently maps a manifold with a variety of resolution [5].” This is based on the intuition that some regions of the manifold consist of a higher density of points than others and that graphs induced from these clusters can provide more information on these regions of the manifold [5].

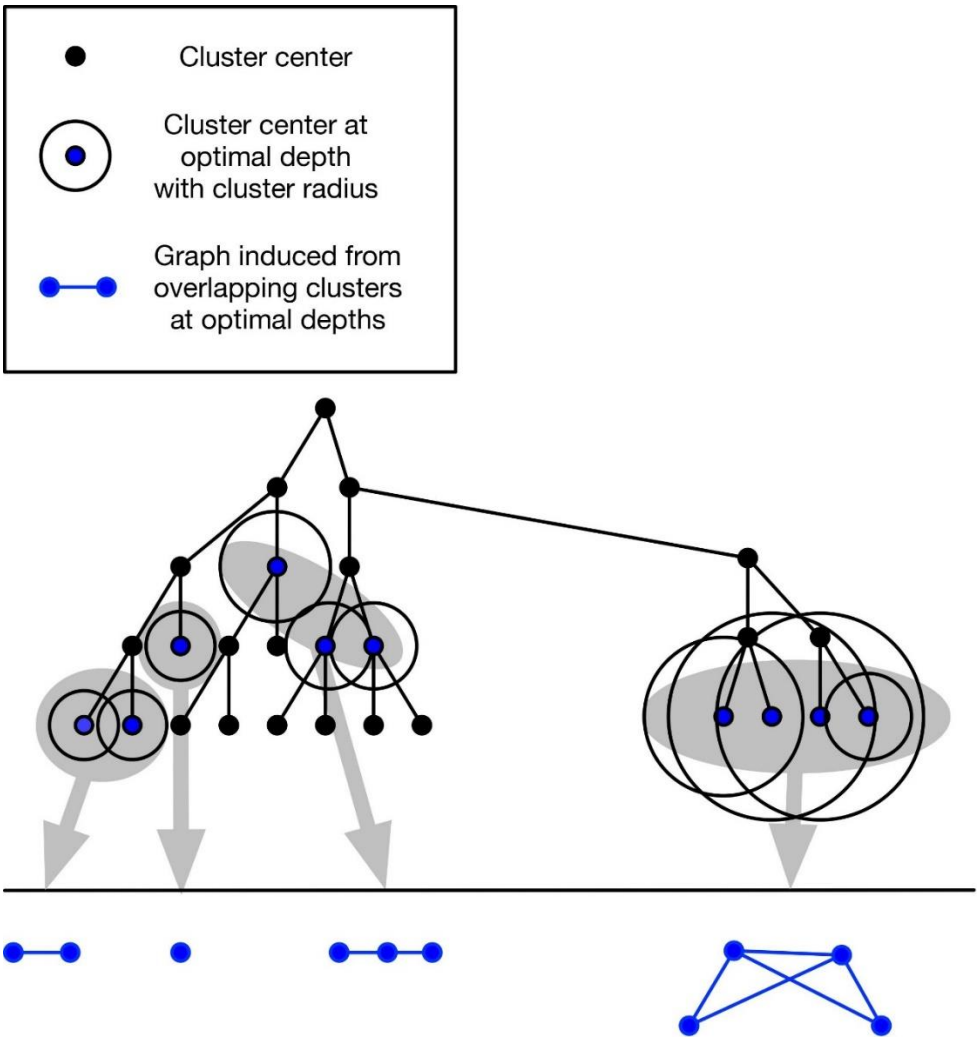


*Figure 1: Blue dots represent clusters that have been selected for the graph. By selecting clusters at a variety of depths, we are ensuring that a manifold without a uniform distribution is properly mapped.*

A CLAM graph exhibits an important invariant in that the sum of the cardinality of all clusters in the graph must be equal to the cardinality of the root cluster in the tree. This essentially means that every datapoint in the dataset will be represented in the graph regardless of how many clusters have been selected. Another notable invariant of the graph is that it will never contain two clusters that are a parent or child of the other [5].

While one could arbitrarily select clusters at various depths while maintaining the previous two invariants, CHAODA offers insight into selecting the “right” clusters that provide a useful representation of the underlying manifold [5]. CHAODA’s focus was on outlier detection in high dimensional datasets, and thus created several scoring functions for clusters that quantify how “good” of a choice a cluster is for the graph.

The algorithm for selecting the clusters involves first sorting the clusters in the tree based on their score. The next step is to take the first cluster from the sorted queue and add it to the graph. The cluster’s ancestors and descendants are then removed from the queue. This process is repeated until the queue is empty. CHAODA also offers a hyper-parameter that allows the user to specify a minimum depth in the tree that a cluster must have. Providing a higher minimum depth will force the algorithm to choose clusters higher in the tree. This can be useful for ensuring that a useful and interesting graph is created. For example, if the minimum depth is zero, the root cluster could be selected as the only vertex in the graph. If the min depth is too high, it could result in many leaf clusters being selected, which would lead to many disjoint graph components.

# CHAPTER 3

## METHODOLOGY

### 3.1 Unity

The first design choice I had to make when I started this project revolved around how I was going to render the visualization. One goal of this thesis is to create an interactive visual representation of the manifold, and one idea I had for that was having an interactive camera that the user could control to move around. I also wanted the ability to have a UI overlayed on the screen for the user to change aspects of the visualization while allowing the clusters on the screen themselves to be clickable objects the user could interact with.

These constraints led to a decision that a game engine should be used to create such a visualization. I initially considered using a game engine written in Rust because that is the language in which CLAM was implemented. However, because Rust is relatively new language, I felt that none of the engines were stable enough or had enough features to develop with. Another option I had was to use a game engine developed in another language and find a way to link the Rust based CLAM code with the engine. Two of the most popular game engines at the time of this writing are Unreal and Unity. While both engines are more than suitable for developing applications, the choice for me boiled down to which engine had better support for interfacing with a Rust library. Currently, Unity has more documentation for using “foreign function interfaces”, which allow a user in a host language to call functions from another language [17].

## 3.2 Foreign Function Interfaces

The Unity game engine uses C# as its scripting language, which means I would need to create a foreign function interface between C# and Rust using C# as the host language. The common ground between C# and Rust is that they can both be easily bound to the C programming language. This means that if I needed to call a Rust function from Unity, I would need to convert the C# variables into a common C representation of that type, pass it to Rust, convert it from Rust's unsafe C type into a safe Rust variable, do any required work in Rust and pass any results back to Unity. This last step, of course potentially involves the same process of converting the Rust type into a c type, and then converting it from C to C#. These Rust functions would also need to be compiled into a dynamically linked library that Unity would link to at runtime.

Some types are trivially converted across the foreign function interface. For example, passing an int from Unity to Rust simply requires writing a Rust function that accepts an i32 as a parameter. In a similar fashion, returning a i32 from a Rust function means storing the output as an int in C#. However, this concept introduces several complications to the codebase.

One such complication that I ran into was that I was not simply calling functions from Rust: I needed to create data structures that were not compatible with C# and have them live beyond the lifetime of the Rust functions. The tutorial I initially followed when developing the source code introduced the concept of a "baton" [18]. The baton is an opaque pointer allocated in Rust and provided to the host (C#) when the library is initialized.

From then on, any operation requested from the host passes the baton back to Rust. Once the host is finished, it passes the baton back to Rust one last time, so the memory is freed [18].

In my case, this “baton” is a struct that stores the CLAM tree, CHAODA graph, and any other data that I need to persist for the lifetime of the program. C# receives this baton and stores it as a “IntPtr” type, meaning that C# has no idea what data type this pointer represents. In Rust, I created a typedef to save some headaches when writing functions. This pointer is defined as seen in figure 1. It is used in figures 2 and 3 when C# calls a foreign Rust function, passing it back the baton with the information Rust needs to return the tree height.

```
pub type InHandlePtr<'a> = Option<&'a mut Handle<'a>>;
```

*Figure 2: The Handle Pointer used to pass CLAM data between Rust and C#.*

```
[DllImport(__DllName, EntryPoint = "tree_height")]
private static extern int tree_height(IntPtr handle);
```

*Figure 3: C# declaration of an external Rust function.*

```
#[no_mangle]
pub unsafe extern "C" fn tree_height(ptr: InHandlePtr) -> i32 {
if let Some(handle) = ptr {
      return handle.tree_height();
  }
}
```

*Figure 4: A Rust function that can be called from C#.*

```
pub struct Handle<'a> {
   tree: Option<Tree<Vec<f32>, f32, DataSetf32>>,
   clam_graph: Option<Graph<'a, f32>>,
   force_directed_graph: Option<ForceDirectedGraph>,
}
```

*Figure 5: The struct that stores all relevant Rust data and needs to persist for the duration of the visualization.*

A common operation that I needed to perform throughout the code base was passing the properties of a Cluster from Rust to C# (or vice versa). To do this, I needed an intermediate C-style struct that could be passed between the two languages, which I called ClusterData. For example, if a user clicked on a cluster to view its properties in the side menu, C# would call a Rust function, passing the name of a cluster as a parameter, and would receive an instance of ClusterData, which stored any relevant information.

While Rust stored the clusters in CLAM, Unity needed a way to store the GameObjects that would be visualized to represent those clusters and would need an identifier so that the GameObject could be associated back to its Rust counterpart. To that end, I created a Unity script called Cluster, which would store an id, position, and color. I could have copied all the properties of a Cluster into Unity, but I felt it would be a waste of memory to duplicate all the values when the datasets could potentially involve millions of clusters.

When I first began my thesis, a cluster's id was created via Huffman Encoding. The naming convention is such that the root cluster is named '1' and each left child would have a '0' appended to its name while each right child has a '1' appended to its name. As the tree was created, its name was

memoized and stored as a string. The authors of CLAM later refactored the struct and removed Huffman encoding as they came up with a more efficient method of naming a cluster based on its offset and cardinality. It would have been easier to use these integers as identifiers instead of the string ids, however this change occurred rather late in my development phase, so I left the id as being stored as a string to avoid breaking changes.

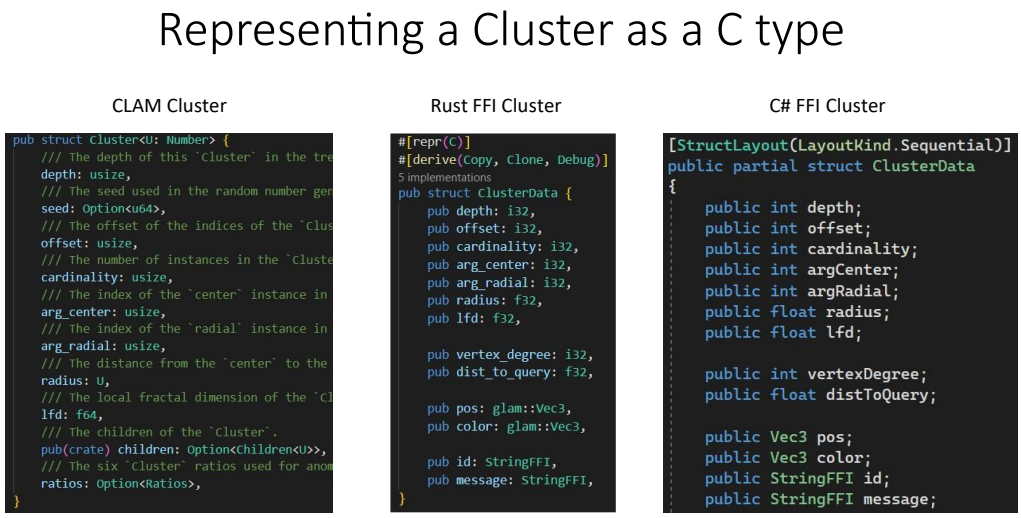


*Figure 6: A comparison of the Cluster struct and the ffi interface struct used to pass data between Rust and C#.*

Passing strings across the foreign function interface led to another challenge because I needed to dynamically allocate unmanaged memory, convert it to a C type and pass it to the other language. This also meant that I was required to manually free the memory down the road. To avoid unnecessary confusion, I decided to introduce an invariant to my code: Rust would always be responsible for allocating and freeing unmanaged strings. To create an instance of this struct in C#, it would pass a safely managed c-string containing a cluster's name as well as a 'out' variable of type

ClusterData to Rust. Rust would fill in any CLAM related variables, allocate the string id and then set the C# variable equal to this instance.

```
[DllImport(__DllName, EntryPoint = "alloc_data")]
extern FFIError alloc_data(IntPtr ptr, string id, out ClusterData data);
```

*Figure 7: The external Rust function used to create ClusterData.*

```
#[no_mangle]
pub unsafe extern "C" fn alloc_data(
    ptr: InHandlePtr,
    id: *const c_char,
    outgoing: Option<&mut ClusterData>,
) -> FFIError {
    if let Some(handle) = ptr {
        let cluster = handle.get_cluster_from_string(id);
        let cluster_data = ClusterData::from_clam(cluster);
        *outgoing = cluster_data;
        FFIError::Ok
    }
}
```

*Figure 8: Rust implementation of function allocating ClusterData.*

Rather than calling this type of function directly, I created a struct called ClusterDataWrapper. Inspired by smart pointers in C++, I made it so the constructor would allocate a ClusterData struct, and the destructor would free the memory automatically when going out of scope. Another addition seen in the image is the FFIError Enum which I use to pass error messages back and forth between Rust and C#.

```
pub struct ClusterDataWrapper {
   data: ClusterData,
}

impl Drop for ClusterDataWrapper {
   fn drop(&mut self) {
      self.data.free_ids();
   }
}
```

*Figure 9: Rust implementation for automatically freeing string memory.*

```
public interface IRustResource {
   void Free();
}

public class RustResourceWrapper<T> where T : struct, IRustResource {
   T m_Data;
   public FFIError result;

   ~RustResourceWrapper() {
      m_Data.Free();
   }
   public T Data {
      get { return m_Data; }
   }
}
```

*Figure 10: C# implementation for automatically freeing string memory (note that this call to free passes the data back to Rust to be freed).*

Another notable aspect of the design of my foreign function interface was the use of callback functions. There are certain cases where I might need to modify the position of every cluster based on data in Rust. To accomplish this, I would create a callback function in C# that accepted a ClusterData as a parameter. In C#, the callback function would behave as normal, meaning I could modify Unity objects and C# objects such as Dictionaries which are normally incompatible with a foreign function interface. Rust could then

traverse through the tree, create a ClusterData object filled with the necessary properties and then invoke the C# callback function.

```
void ColorByCardinality(ref Clam.FFI.ClusterData nodeData) {
   tree.GetValue(nodeData.id, out var cluster);
   float ratio = 1.0f - nodeData.cardinality / tree.cardinality;
   cluster.GetComponent<Cluster>().SetColor(new Color(ratio));
}
```

*Figure 11: A callback function passed from C# to Rust that sets a cluster's color according to its cardinality.*

```
public delegate void NodeVisitor(ref Clam.FFI.ClusterData data);
```

*Figure 12: A typedef of the function callback signature in C#.*

```
type CBFnNodeVisitor = extern "C" fn(&ClusterData) -> ();
```

*Figure 13: A typedef of the callback function used in Rust.*

```
fn for_each_dft(root: &Cluster>, node_visitor: CBFnNodeVisitor) {
      if let Some(cluster) = root {
      let ffi_cluster = ClusterDataWrapper::from_cluster(cluster);
      node_visitor(ffi_cluster.data());

      for_each_dft(root.left, node_visitor);
      for_each_dft(root.right, node_visitor);
   }
}
```

*Figure 14: The Rust function that accepts a higher-order function to perform on clusters.*

The design of this interface allows me to write code that transfers data between Rust and C# without having to directly interface with any of the unsafe C code in the middle. Any unsafe memory allocations are automatically freed, any unsafe types are converted to safe types before I start working on them. I essentially designed this so that I wouldn't really need to worry about the unsafe nature of foreign function interfaces when I was implementing the

backend code. On top of that, the design of being able to use higher order functions while iterating over clusters in the tree allowed for more generic re-usable code.

## 3.3 The Tree

While Rust stored Clusters in the tree-like structure described in the literature review, C# stored the companion cluster objects in a Dictionary where the key is the name of the cluster (a string), and the value is the Unity Object representing the cluster.

This setup of copying the tree into a C# dictionary leads to a potential memory bottleneck where I could end up duplicating all the memory stored in Rust. I used two methods to solve this. One is that I made the attributes of a Rust cluster and a Unity structure mutually exclusive. Rust would store the clusters as described in the literature review section while Unity would store the name of a cluster, its location, and its color. This at least prevents issues of duplicate data as with large datasets, even duplicating integers and floats could be costly if there are millions of them.

The second method that I used to optimize the memory was based on the observation that there is little need to store every cluster in the tree in Unity at any given time and instead I decided to only create Unity objects for clusters that the user cares about. To that end, I treated Unity's dictionary like a cache and Rust as the server. I initially only store the first few levels of the Unity tree in the C# dictionary. If a user requests to view another level in the tree, I check if the clusters on that level exist in Unity yet. If they do, I simply make the

clusters and any corresponding edges visible. If not, I add a cluster to the C# dictionary and place it on the screen according to layout calculations that will be covered in the Reingold Tilford section. This behaves in a similar manner for when the graph is constructed and if a user attempts to view the subtree of any clusters in the graph.

### 3.4 Implementing the Force Directed Graph

Calculating the forces and applying them to each cluster is a costly operation and can be a bottleneck during the physics simulation. Unity has built in physics systems that could be leveraged here, but I chose to implement the graph in Rust because of how closely coupled it is with the CLAM graph. A naïve approach to this would involve Unity calling a Rust function that computes the forces and waits for it to finish. However, this freezes the entire application while it waits on Rust. To resolve this, I implemented the producer consumer idiom.

Rust's physics simulation would run on a worker thread, calculating the forces to be applied on the edges. Once it was finished calculating, it would go to sleep and wait to be awoken. I then modified the update function of a Unity object so that it would attempt to have Rust take the finished calculations and update the Unity objects. If the calculations were not complete, the function would end, and the objects would not be updated during that frame. However, if the calculations were complete, CLAM would iterate through each cluster in the graph and use callback functions to update the respective Unity objects'

positions in 3D space. This function would then send a signal to the worker thread, waking it up to work on its next iteration.

In the cases of large graphs, performance bottlenecks can still be noticed as the clusters won't be moving each frame. However, it is still better than the naïve method because it reduces the amount of time that Unity is waiting on Rust.

### 3.5 Edge Drawing

Unity provides a line renderer component for drawing lines between two or more points that I initially used to draw the edges between clusters in the tree. However, I noticed a performance bottleneck because the line renderer component is not well optimized for handling thousands of lines. To optimize this, I introduced an index and vertex buffer on a mesh and manually drew the lines.

One aspect of the improvement in performance came from the reduction in the number of game objects to update. Five thousand objects would take five thousand update and draw calls during the main loop. Creating one mesh object to draw all the lines only takes one update and draw call. Another aspect of the optimization came from the reduction in the number of vertices being placed on the screen. The purpose of an index buffer is to re-use existing vertices rather than draw duplicates on screen.

Take for example, a graph consisting of four vertices where each vertex has an edge connecting it to each other vertex. This leads to a graph with six edges. Drawing the edges naively would result in placing two vertices on

screen for each line (12 vertices). However, using an index buffer to re-use vertices, we can reduce the number of vertices drawn on screen to just four.

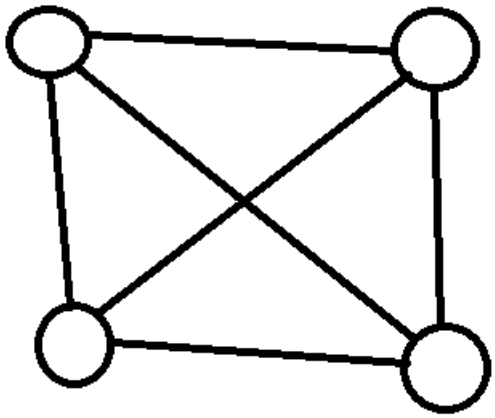

*Figure 15: Example of a shape that can be drawn by reusing vertices with an index buffer.*

To summarize, this means that drawing the graph requires drawing a number of vertices equal to the number of clusters that have an edge, rather than the number of edges x 2. Changing to this method led to an impressive boost of performance seen in table one.

| Line Renderer | | | Mesh Renderer | |
|---|---|---|---|---|
| num_lines | fps | | num_lines | fps |
| 1000 | 200 | | 1000 | 400 |
| 3000 | 90 | | 3000 | 300 |
| 5000 | 70 | | 5000 | 300 |
| 10000 | 40 | | 10000 | 280 |

*Table 1: A comparison of the performance of Unity line renderer components vs A mesh renderer with index and vertex buffers*

3.6 Refactoring CLAM and CHAODA

Another factor that went into the implementation of this thesis was that the CHAODA paper had originally been published with a codebase written in Python. Most of the code had been ported to Rust, but it was outdated and no longer compatible with the latest version of CLAM. In addition, parts of CLAM such as the Ratios, which are required to score clusters were not implemented. To push my changes to the Clam repository, I also needed the code to pass several quality checks setup by the research group.

In some cases, the refactoring was as simple as adding documentation to a function or struct (although that does add up when there are hundreds of functions that need documentation). Other times I wrote unit tests for the Graph and CHAODA modules, which led to the discovery of some bugs, including an improper implementation of standard deviation. I also needed to update the types that the Graph struct was using so that it matched the latest version of CLAM. This could mean adding lifetime specifiers or changing names of variables or functions.

Within the CLAM codebase itself I needed to add the ratios calculations as an optional function that could be called when building the tree. This would memoize the ratio values of each cluster compared with its parent. The ratios play an important role in how CHAODA selects optimal clusters to be in the graph. Overall, this took a couple of months before I had the changes all merged into their codebase and ready for use with my visualization.

## 3.7 Displaying the Tree

The primary purpose of my thesis is to view an induced graph representing the underlying structure of a manifold in a reduced dimension. However, I felt it important to visualize the CLAM tree in addition to the graph for several reasons. The first was simply because when I first started this project, I was new to Unity, Rust, CLAM, and the concept of foreign function interfaces. I also did not fully grasp how the graph would work yet. I did, on the other hand, understand what a tree-like structure was. So, I taught myself about all these topics by creating a visual representation of the tree before I started any work on the graph. After I finished creating the tree, I chose to include it in my final visualization application because it helps provide insight into the underlying structure of the data as well as provide insight into CLAM.

As noted by the authors of CLAM, the tree is typically unbalanced and left leaning because they state that data is not always evenly distributed along the manifold. In addition, when creating the graph, I provide an intermediate step where, based on the hyperparameters specified by the user, the clusters that have been chosen to build the graph are highlighted for the user. This gives them a visual representation of how deep in the tree their graph is. It also helps users distinguish the difference between the purpose of the graph and the tree altogether.

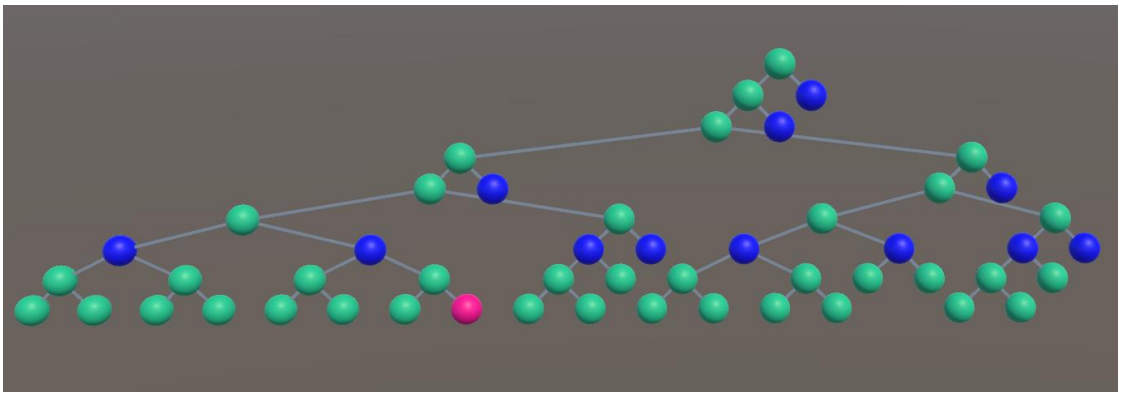

*Figure 16: An example of the Reingold Tilford tree layout. Blue clusters have been selected to create a graph from this tree.*

### 3.7.1 The Reingold Tilford Layout

A challenging aspect of visualizing the tree is that trees of certain heights become increasingly difficult to visualize effectively. In fact, for large enough trees, the spacing required to place the clusters in a naïve manner could result in only the root being visible as the other clusters were pushed off into space. This is also affected by the fact that the trees are typically left-leaning and my initial spacing algorithm did not account for the fact that the left side might need more spacing than the right side.

To rectify this, I researched several tree visualization algorithms and settled on the Reingold-Tilford algorithm which aimed to preserve several invariants. First was that nodes at the same depth in the tree should be drawn in the same vertical level in the tree. Next was that a left child should be drawn to the left of its parent, and the right child to the right of the parent. The parent should be centered over the children, and a subtree should be drawn the same way regardless of where it appears in the tree [19]. In addition to the properties, Reingold and Tilford aimed to achieve this while minimizing the

width of the tree. However, I found that even with this algorithm designed to minimize width of the tree, large datasets were still not easy to view.

I redesigned the algorithm to have a specified starting cluster, not just the root of the tree, and I added an additional parameter called max depth as stopping criteria. The latter allowed the user to only space clusters as if the tree was a specified height, rather than its actual height. This allowed clusters close to the root to initially be placed more closely together. As the user requests to view deeper in the tree, the max depth parameter is increased and the layout of the tree is recalculated with this new height, behaving in much the same way as a dynamically resizing array. Changing the algorithm to work from any cluster rather than just the root comes in handy when visualizing the subtrees of clusters in a graph. The results of this implementation can be seen in Figure 17, Figure 18, and Figure 19 below.

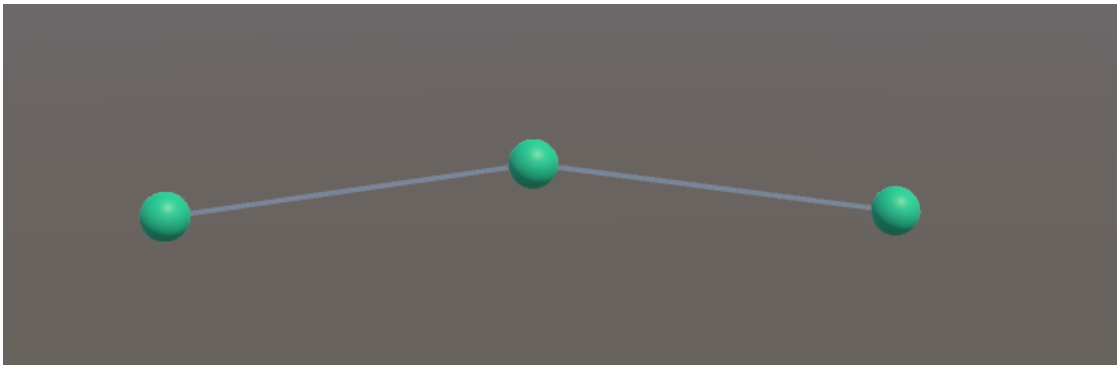

*Figure 17: A tree with a max height of 30 being displayed as though its max height was 2.*

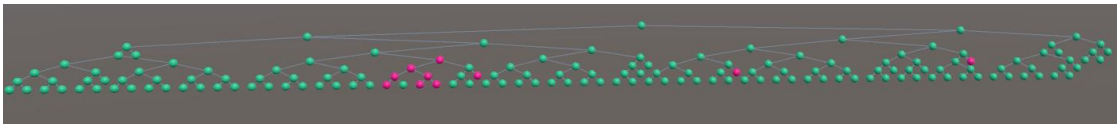

*Figure 18: A tree with max height 30 being displayed as though its max height was 8.*

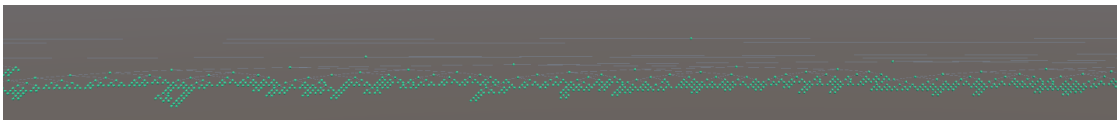

*Figure 19: A tree with max height 30 being displayed. The tree is too wide to fit on screen even with the use of the Reingold Tilford Algorithm.*

## 3.8 Cluster Properties

An important aspect of this thesis was to provide an interactive visual to the user. As such, the user can click on clusters and have their properties displayed on screen. The current UI design behaves much like the project TensorFlow visualization where you can click on a cluster to view its properties. However, the hope of using this clustered manifold mapping technique is that providing information about the cluster properties will provide additional information about the manifold. For example, the local fractal dimension of a cluster provides insight into the actual embedding space of the cluster. Other properties such as cardinality and radius can be used to determine how densely packed the cluster is. In addition, the offset combined with the cardinality allows the user to know which indices in the dataset are represented by that cluster.

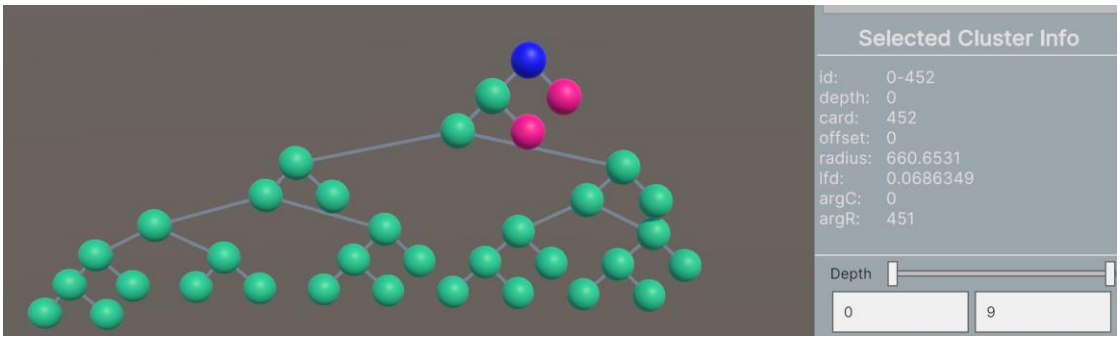


*Figure 20: The blue cluster has been selected by the user, displaying its properties in the menu on the right.*

### 3.9 Coloring Clusters

This visualization tool allows the user to color clusters based on several of the clusters' properties including cardinality, local fractal dimension, radius, label, depth, and vertex degree. Currently, all coloring properties besides by label are in greyscale and are normalized to the dataset. For example, if coloring by cardinality, a white cluster has a low cardinality compared to a cluster colored dark grey or black. Future HCI experiments could be conducted to research how users would prefer the color scheme be represented.

I initially created this visualization using the anomaly detection datasets used in the CHAODA paper, which only had two labels: inlier or outlier. As such, I setup the coloring scheme so that a red cluster would signify an outlier and a green cluster an inlier. I also scaled the color gradient based on the entropy of the cluster. So, the red channel was calculated as the number of outliers divided by the cardinality of the cluster and the green channel was calculated as the number of inliers divided the cardinality of the cluster. This would lead to clusters with a mix of inlier and outlier data being a yellowish color.

Later, I added functionality to support up to ten labels. However, I had to drop the shifting gradient idea because it could result in misleading colors of clusters. For example, if I assign a unique cluster for each label and those colors include yellow, green, and red, then a cluster made up of half green, half red could be the same color as a cluster with the yellow label. One other

notable aspect of the coloring scheme is that it is designed to be colorblind friendly, using a tool developed by Adobe that warns if certain colors clash in certain types of color blindness. The colors are still setup so that if using the anomaly detection datasets, green are inliers and red are outliers, as seen below. Coloring clusters in the tree and graph based on their properties allows the user to gain a better understanding of how clusters work while also portraying important properties of the tree and graph.

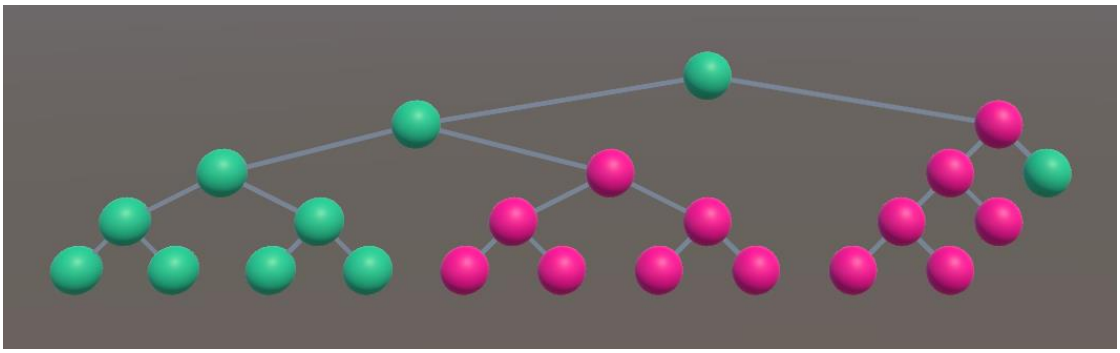

*Figure 21: Coloring a tree by label where green is inlier and red is outlier.*

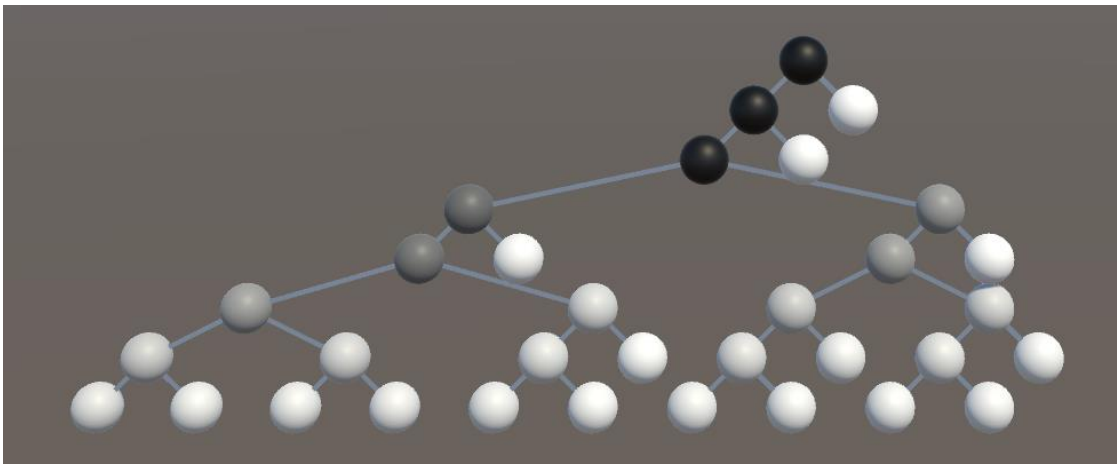

*Figure 22: Coloring a tree by cardinality. Darker colors mean a higher value.*

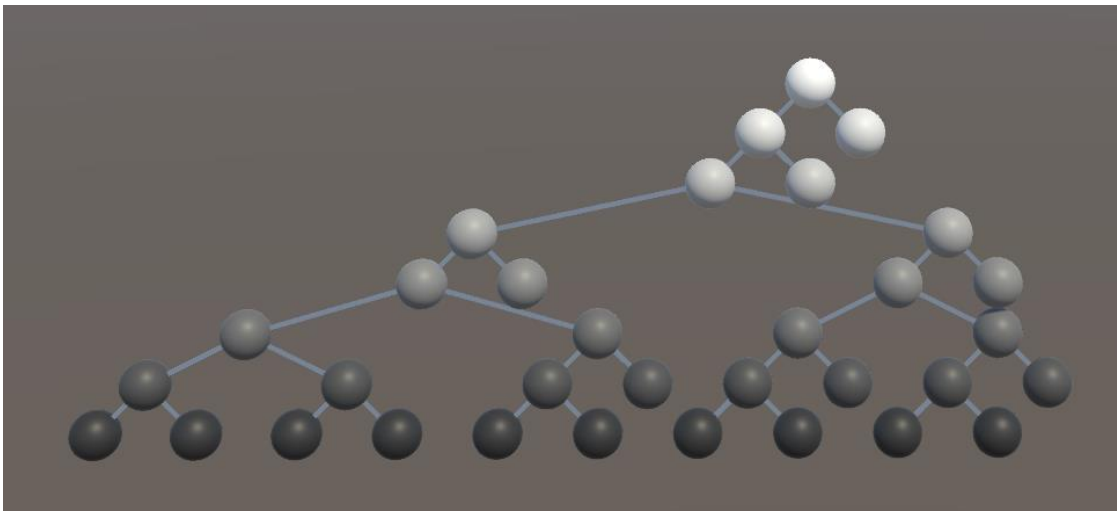

*Figure 23: Coloring a tree by depth.*

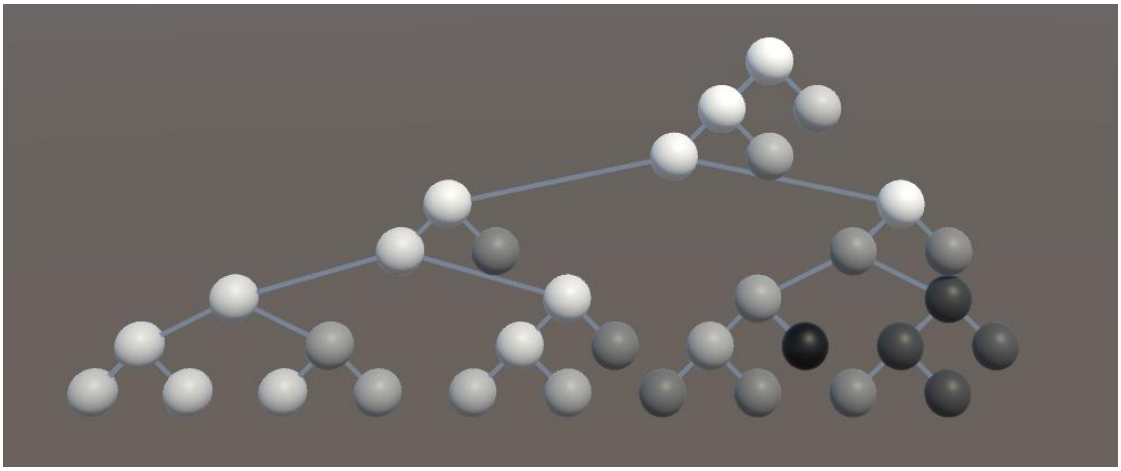

*Figure 24: Coloring a tree by local fractal dimension.*

## 3.10 Selecting Clusters with CHAODA

As seen in the CHAODA review in the previous section, an important consideration for building the graph is how the clusters are selected. To summarize, clusters must be selected from the tree such that every datapoint in the original dataset is represented in the graph while avoiding duplicate datapoints. One could arbitrarily select clusters that meet these criteria but there is a chance they could end up with a graph containing too few or too

many components. To avoid the first problem, the user interface suggests to the user that they build the graph with a min depth of 4 by default.

By increasing the min depth parameter, the user is guaranteeing that their graph will consist of more clusters. If the user goes too far, they could potentially end up with all leaf clusters chosen, which would lead to a disjoint graph with few to no edges. One note about the algorithm is that it will only ignore non-leaf clusters below the minimum depth parameter. For example, in Figure 25, the right child of the root is a leaf cluster and is chosen for the graph despite being lower than the minimum threshold. If this was not considered, we would break the invariant that each point in the dataset be represented in the graph.

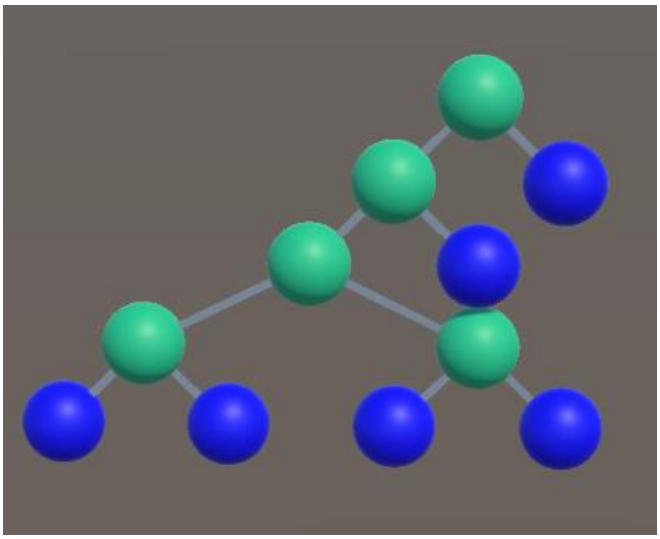

*Figure 25: The blue clusters have been selected to build a graph with minimum depth 4 from a tree with a height of 4. Leaf clusters with a depth less than min_depth are still selected in order to maintain the invariant that the graph represents the entire dataset.*

CHAODA selects clusters based on scoring functions produced through a meta machine learning model. The scoring functions accept the “ratio” of a cluster as a parameter and return a score representing how good of a choice a cluster is for the graph. A ratio of the cluster is an array of six floats consisting of its cardinality, radius, and local fractal dimension divided by its parent’s corresponding values as well as the next exponential moving average of each value. The next exponential moving average is used to place a greater weight on more recent values while still taking older values into account.

One issue I ran into with CHAODA’s cluster selection was dealing with tiebreakers. For example, if clusters had the same score, the order that the clusters appeared in the queue was not stable between runs. This means that you could give the program the same input twice and receive different outputs. To address this, I added custom comparators to CHAODA’s codebase that would rank clusters by highest score, lowest offset, highest cardinality. The intention here is provide a heavier bias towards clusters close to the root as the deeper in the tree you go, the higher the offset and lower the cardinality. If clusters deeper in the tree were ranked higher, it would be difficult for the user to intentionally create a smaller graph. However, if the bias is towards clusters closer to the root, the user can just increase the min depth parameter if they want a more complex graph. An example of this can be seen in Figure 26 and Figure 27, where the min_depth value is increased, resulting in more clusters being selected. Figure 28, Figure 29, Figure 30, and Figure 31 also show how the graph becomes more populated as the min_depth parameter is increased.

I also provided an intermediate step for the user where they could preview the graph before running the physics simulation. Clusters selected for the graph would be highlighted blue in the tree, giving the user an idea of how deep in the tree their cluster selection is. In addition, I display properties such as the number of graph components, number of clusters, and number of edges present in the tree. This allows the user to experiment with different min_depth values and preview properties of the graph

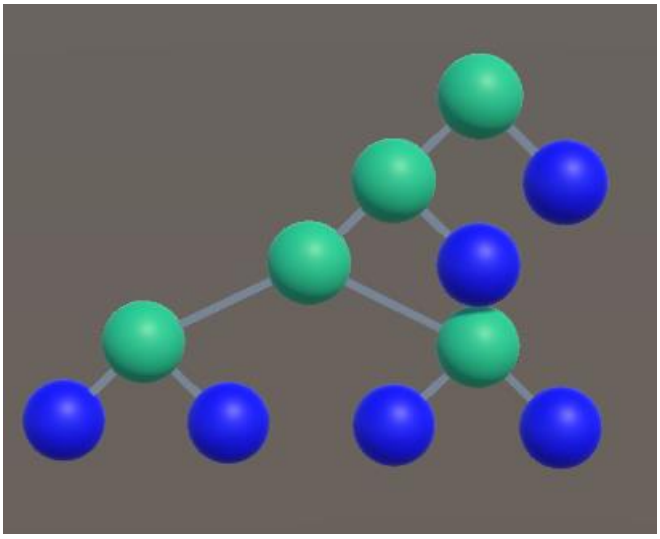

*Figure 26: Clusters selected for a graph with min depth of 4.*

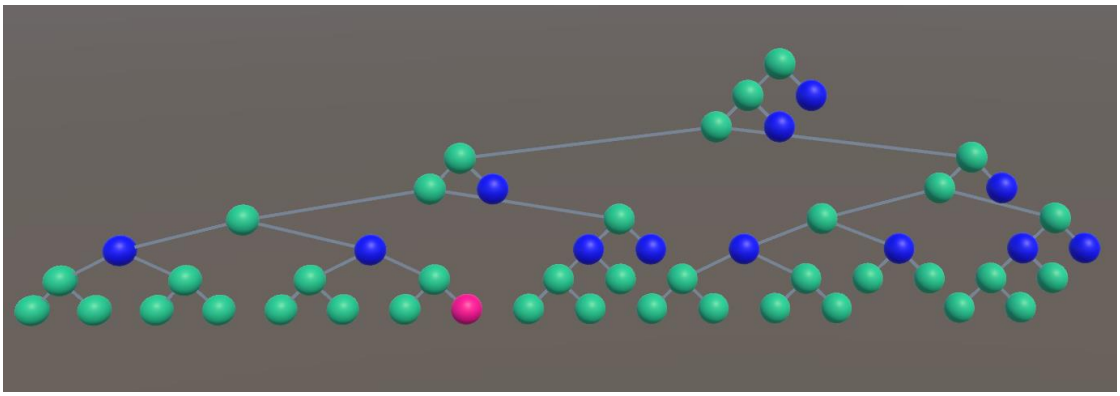

*Figure 27: Clusters selected for a graph with min depth 6 from a tree with height 11.*

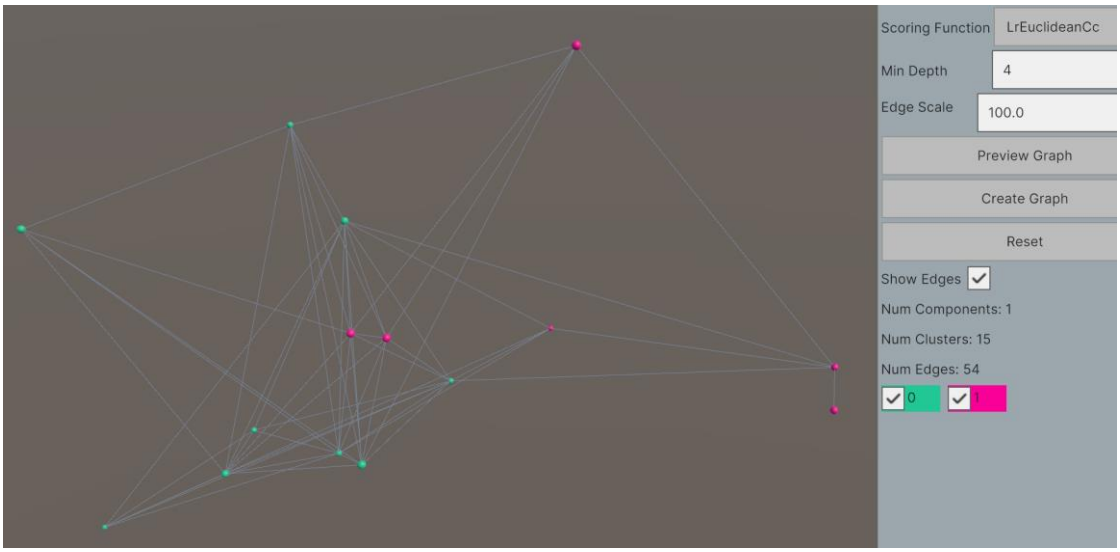


*Figure 28:  A graph built from the annthyroid dataset with min depth 4.*

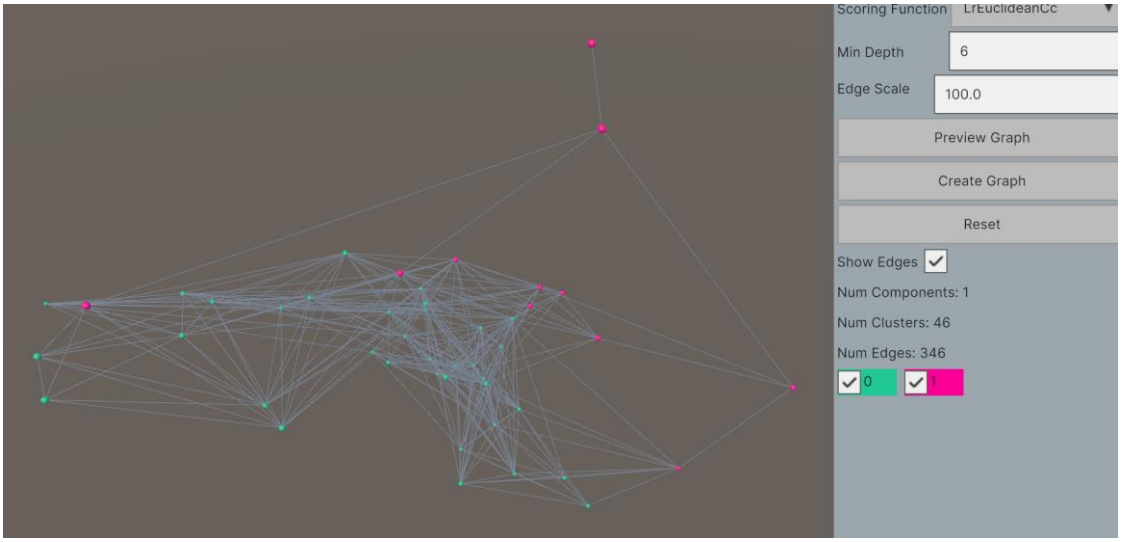


*Figure 29: A graph built from the annthyroid dataset with min depth 6.*

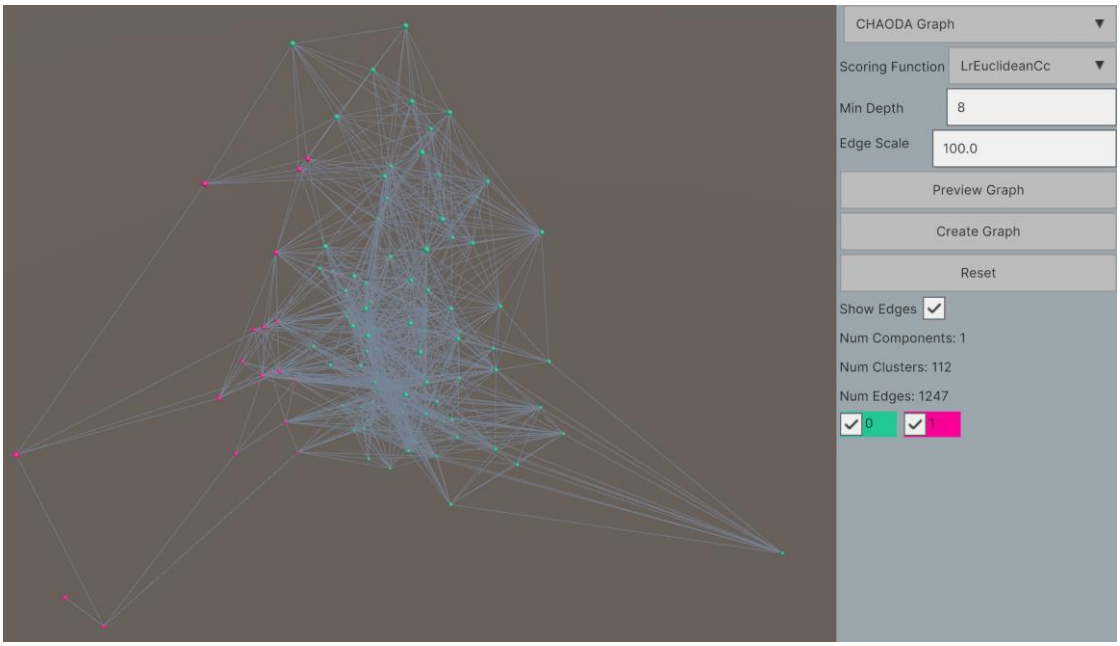


*Figure 30: A graph built from the annthyroid dataset with min depth 8.*

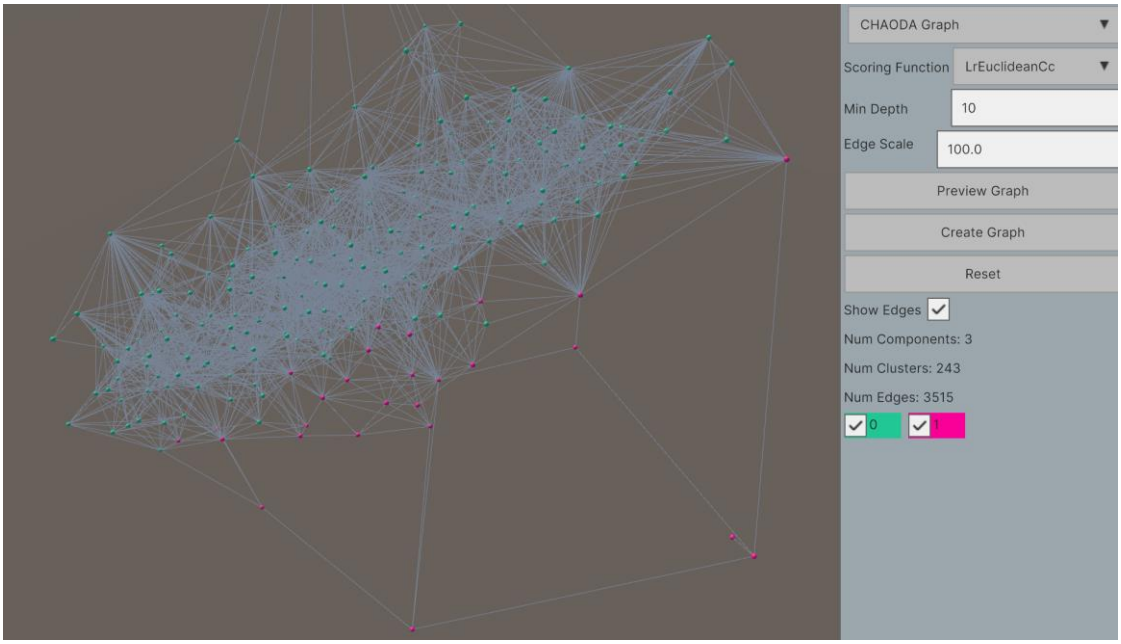


*Figure 31: A graph built from annthyroid with min depth 10.*

### 3.11 Detecting Edges

Common manifold learning algorithms such as t-SNE and UMAP use k-nearest neighbors to construct their graph. In contrast, this visualization uses

clustered manifold mapping. By selecting clusters from the tree, the algorithm is reducing the number of individual datapoints that need be considered while building the graph. This is an important aspect of manifold learning as one the key steps described in the introduction is that the dataset should be distilled to a subset of its data that represents key features. The nature of the CLAM tree groups alike datapoints into clusters. However, we still need to create edges between these clusters. Recall that edges represent some sort of similarity between datapoints in manifold learning. The formula used to detect edges between clusters is that the pairwise distance between them is less than or equal to the sum of their radii and the target length of the edge between the clusters is simply their pairwise distance. The current implementation compares each cluster in the graph to each other cluster in the graph and has a time complexity of $O(|C|^2)$. Future work could investigate a method of optimizing the detection, which would speed up the building process of the graph.

A problem that appears in some graphs is that the sheer number of edges can obscure the relationships between the clusters. The ability to hide edges as seen in figure 32 can provide the user the choice of if they want to look at edges.

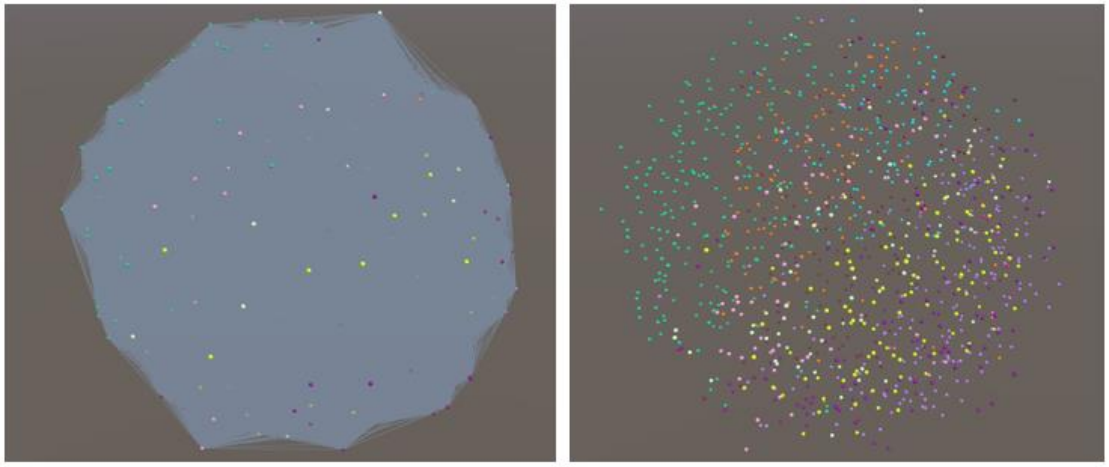

*Figure 32: As seen with the MNIST dataset, it is possible for a graph to have too many edges, cluttering the image and detracting from the information. Hiding edges is an easy fix for this.*

### 3.12 Balancing Local and Global Forces

The next steps follow the similar algorithms seen in the sections on UMAP, t-SNE, and force directed graphs. The initial positions of the clusters are randomly initialized. The graph is then iterated over a user-specified number of times and forces are applied each iteration. For each edge in the graph, forces are computed that will attract clusters together if their physical distance is greater than their metric distance. If the clusters are too close, a repulsive force will be applied instead.

These forces represent the local structure of the geometry of the manifold. The global geometry consists of the relationship between disjoint components in the graph. Other manifold learning algorithms apply some mixture of repulsive forces to avoid crowding and attractive forces to keep datapoints in frame. In this force directed graph, I aim to find an ideal distance between the components themselves. This means that each component is given edges that connect it to each other component. The method choosing clusters to

represent its disjoint component could be a topic of further research, however I reason that the clusters with a high connectivity would be a viable choice. To that end, I sort the clusters in a disjoint component based on their vertex degree and select (up to) the highest three ranked clusters. I specify “up to” because there are some cases when a disjoint component consists of only one or two clusters. This sampling comes from the concern of time complexity of the force directed algorithm.

This now leads us to the tradeoff between local and global structure of the graph. By estimating a target distance between disjoint components, the global structure of the graph should increase in quality. However, by applying forces between the disjoint components, I am disrupting the local representation of each component. One proposal for avoiding this would be to treat each component as a static object and apply forces between the object instead of the individual clusters within it. This would mean that the components themselves would be translated or rotated in respect to one another but the clusters within each component would not have their respective distances altered. This approach seems promising but requires a significant amount of work involving creating Unity objects for each component and applying force and torque between the objects based on forces applied from the location of one cluster to another. This implementation should be investigated in future work to preserve the local layout while optimizing the global layout however due to time constraints and the initial accuracy

benchmarks of the application looking promising, I have opted to leave the implementation as-is for now.

# CHAPTER 4

## FINDINGS

### 4.1 Examples of Created Graphs

The figures below are examples of graphs created using the CLAM visualization tool with the http anomaly detection dataset. In Figure 33, a broader perspective reveals two large components, a small component, and many tiny disjoint components. Figure 35 zooms in on the large component on the left, showcasing clusters packed together into a rope-like structure with a knot in the center. Meanwhile, Figure 34 focuses on the large component on the right, which seems to flatten into a more 2D square shape. The red clusters signify outliers and are notably visible as small disjoint components outside of the larger inlier sections. In Figure 39, I examine an inlier cluster that has two edges connecting to outlier clusters by expanding its subtree. It becomes clear that the edges exist because while the cluster has outlier data in its children.

Figure 38 shows a graph created using the Satellite dataset. I chose to make the edges invisible for this image because they blocked the view of the clusters. The graph shows that the inlier and outlier clusters separate into two different sections with some overlap in the middle. This shows that the graphs created with anomaly detection datasets can be used to visualize the relationship between inlier and outlier clusters.

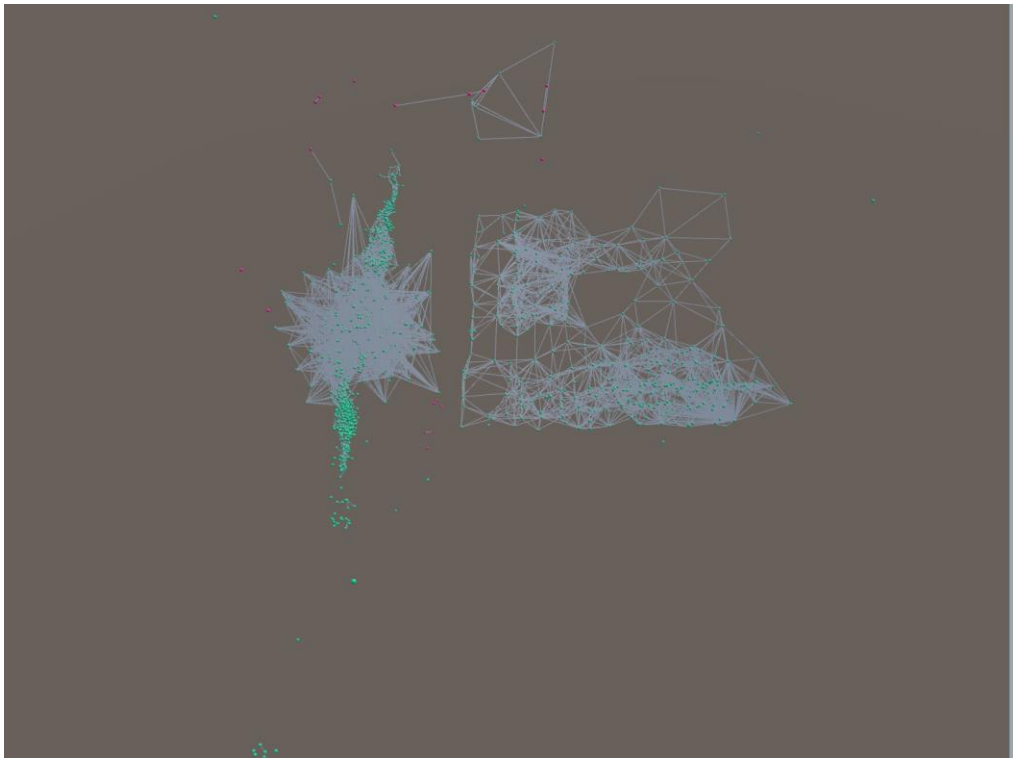

*Figure 33: An induced graph created with the http dataset.*

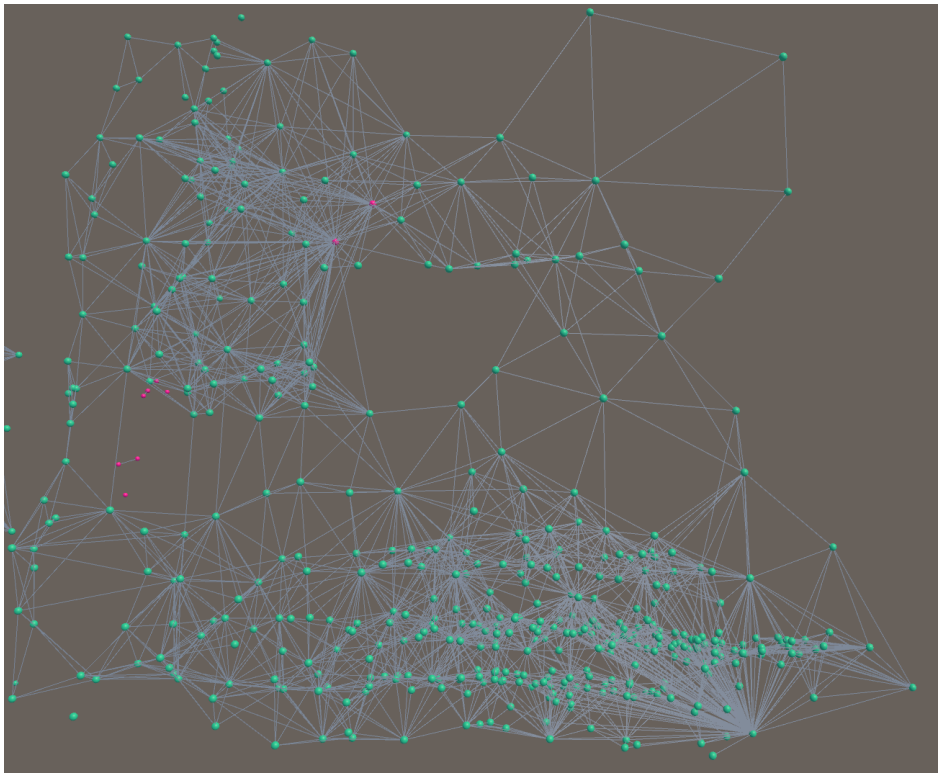

*Figure 34: A close-up of the 2D-like graph component found within the http graph.*

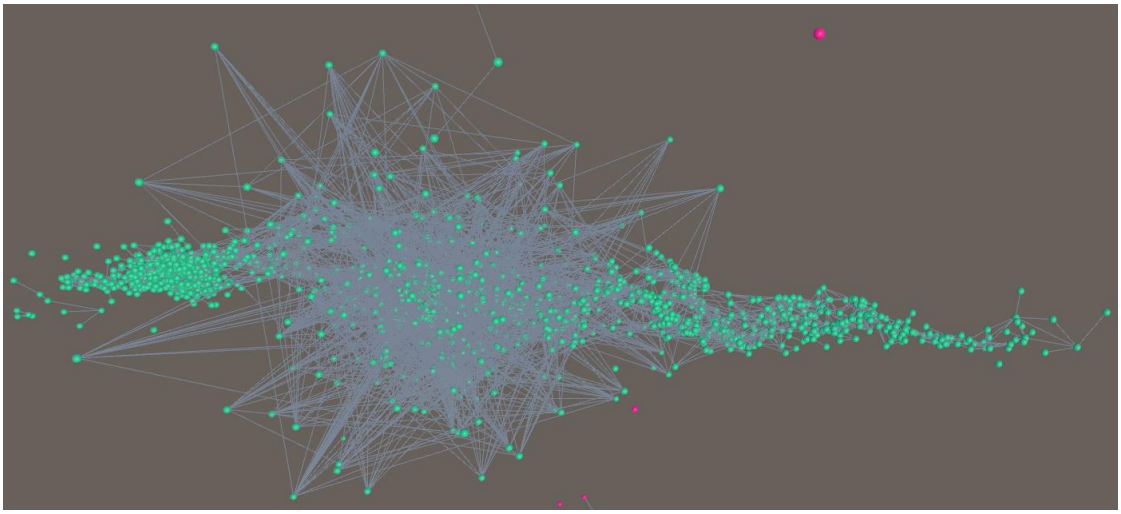

*Figure 35: A close-up of the second large component in the http dataset. It looks like a straight line with a clot in the center.*

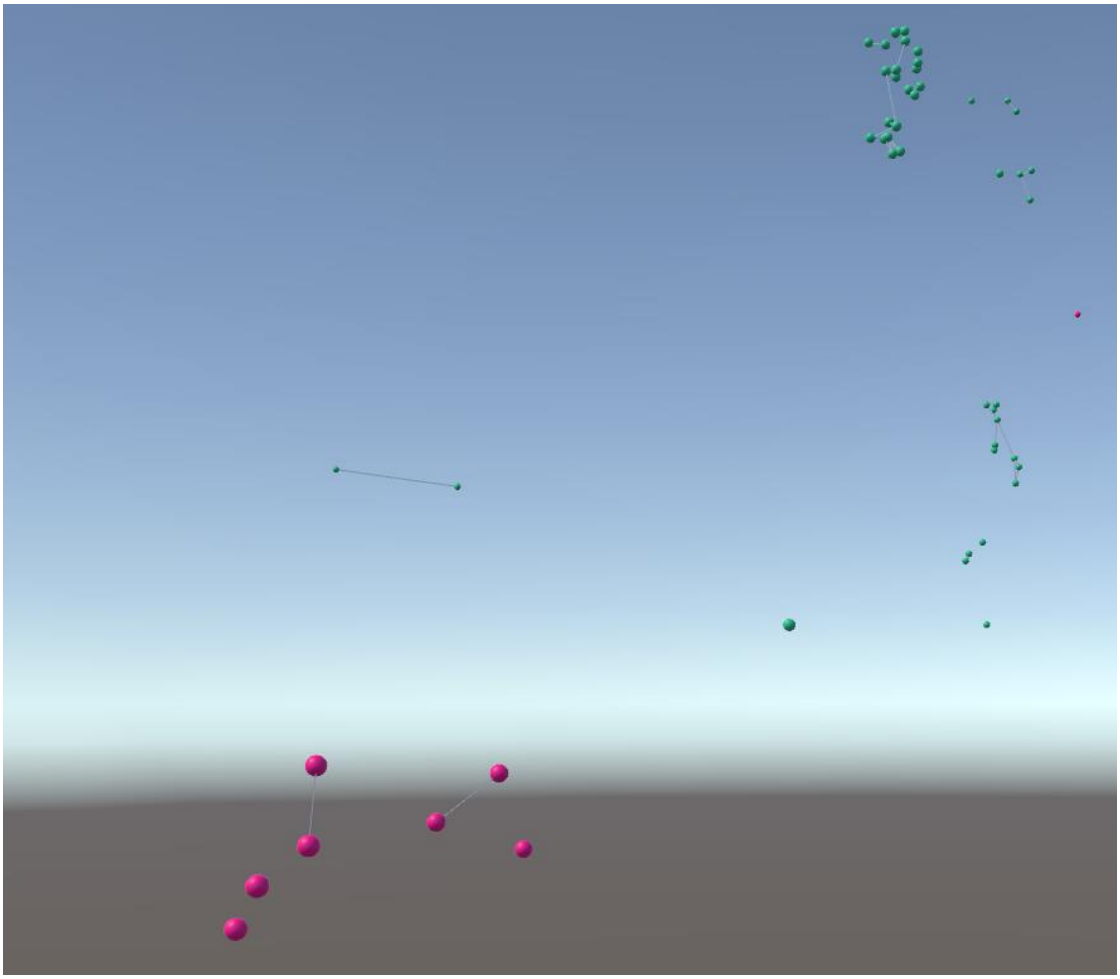

*Figure 36: A closeup of some of the smaller disjoint components in the http graph.*

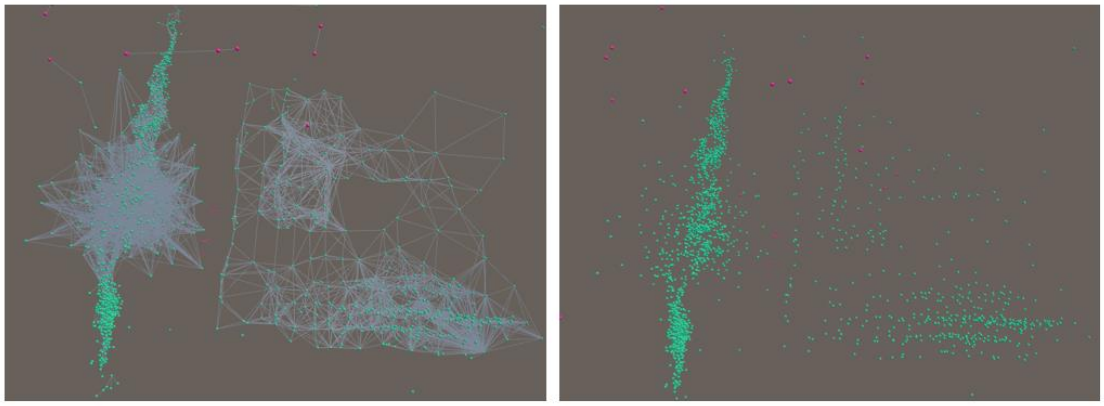

*Figure 37: The http dataset with edges (left) and without edges (right).*

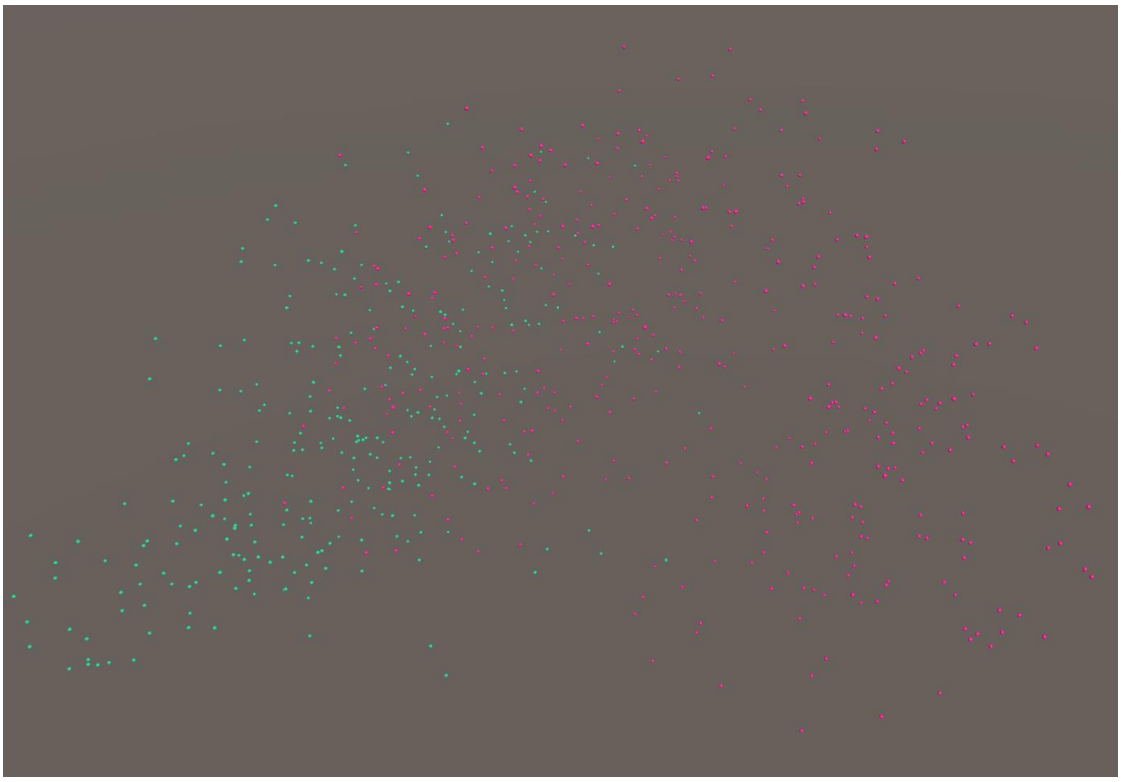

*Figure 38: The satellite dataset produces a graph where inlier clusters are mostly located on the left, outliers on the right, and a small mixture in the middle.*

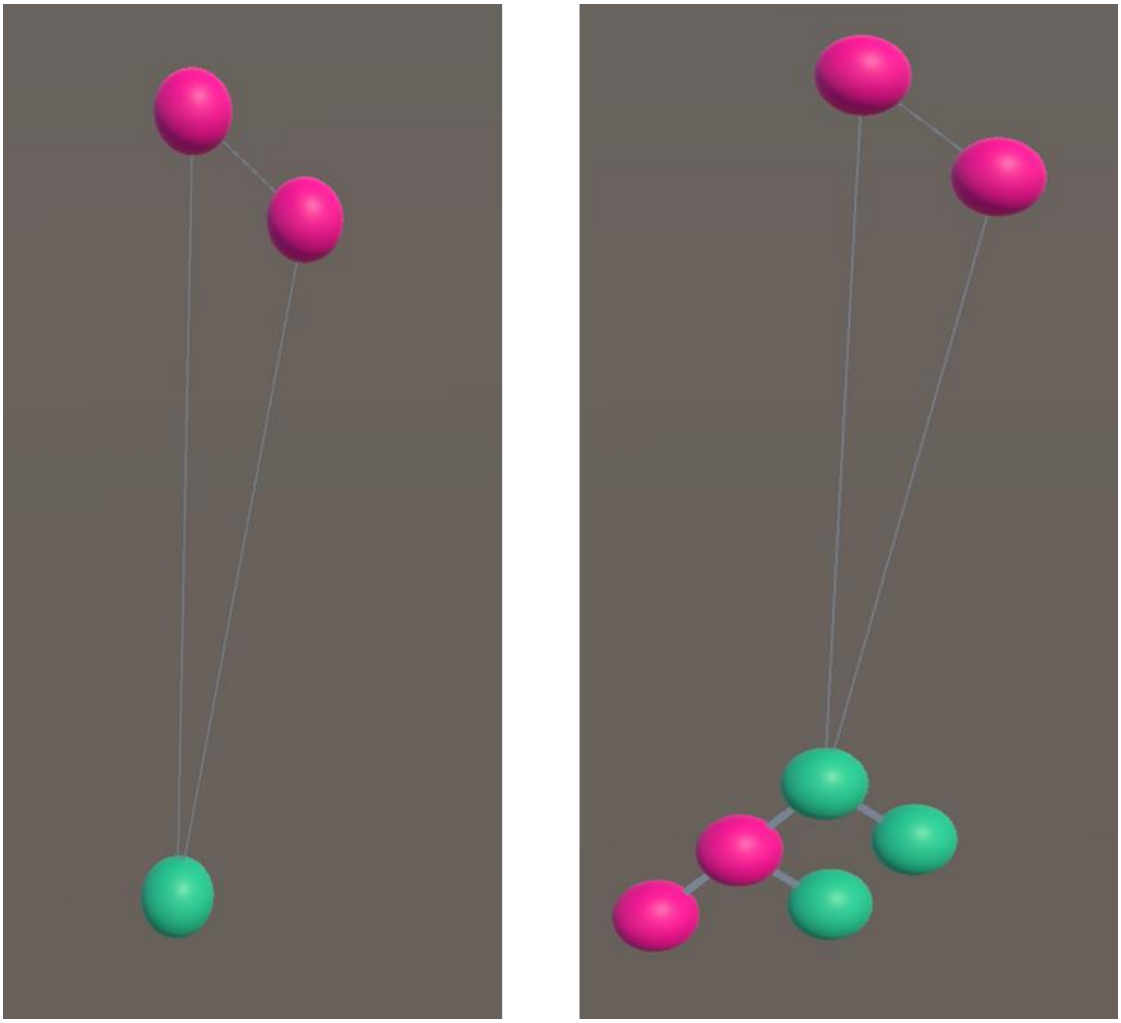

*Figure 39: A small component within a graph has two outliers connected with an inlier (left). By displaying the inlier cluster's subtree (right), we can see that it contains an outlier cluster, which could explain why it has an edge connecting to an outlier.*

## 4.2 Comparisons to Existing Methods

### 4.2.1 A visual Comparison

In the UMAP paper, the authors show how their visualization compares to t-SNE's visualization of the MNIST dataset (see Figure 40). UMAP's results showed that 0 and 1 were on opposite sides of the map while 3,5, and 8 were grouped together and 4,7, and 9 were grouped together. Their visualization does a better job of portraying the global structure than t-SNE's because it not

only groups alike datapoints together, but it gives a better representation of the relationship between unrelated data points.

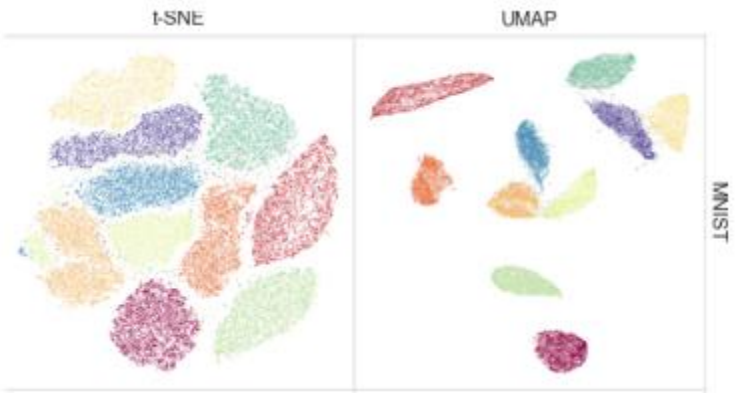


*Figure 40: A Qualitative comparison of t-SNE and UMAP (Melville, 2020).*

Examining the results produced by my visualization (seen in *Figure 41*) is more difficult in 3D space because the result is a mostly hollow sphere that creates misleading results when viewed in 2D form. To compare the results with UMAP and t-SNE, I disable the rendering of certain digits and change viewing angles to highlight the relationships between the digits I want to focus on. In *Figure 42*, I highlight that the 0 and 1 digits are located on opposite sides of the sphere, matching how the digits are located in the UMAP and t-SNE visualizations. In *Figure 43*, I show that digits 3,5, and 8 are located on the left side of the sphere, leaving the digit 7 on the right as a reference frame. In *Figure 44*, I examine the relationship between 4, 7, and 9 by showing how they are located on the right side of the sphere, leaving digit 3 on the left as a reference frame. I included 3D visualizations of the MNIST dataset created by UMAP to show a more direct comparison of the results as seen in figures *Figure 45* and *Figure 46*.

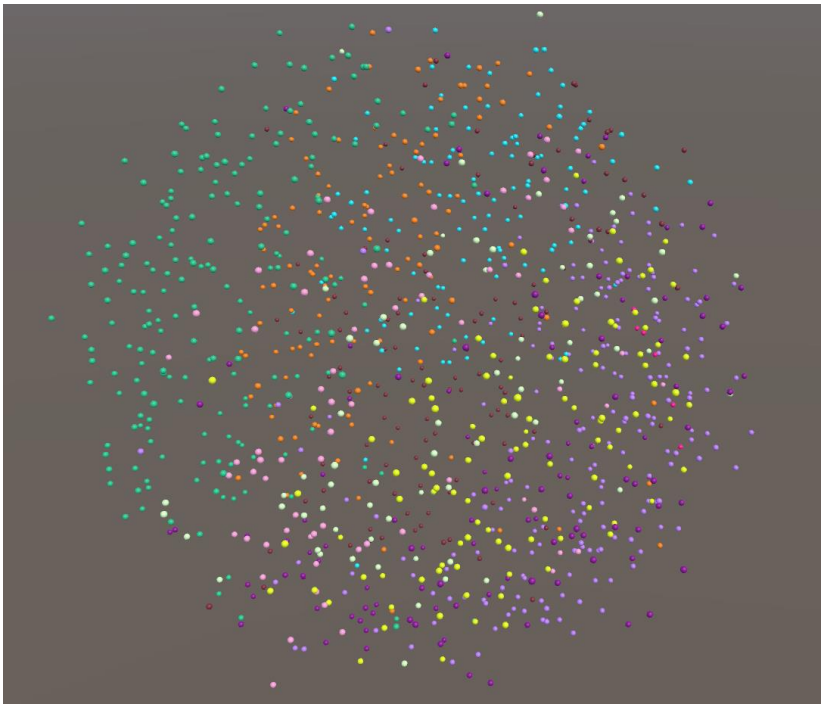

*Figure 41: The graph of the MNIST dataset produced by my visualization.*

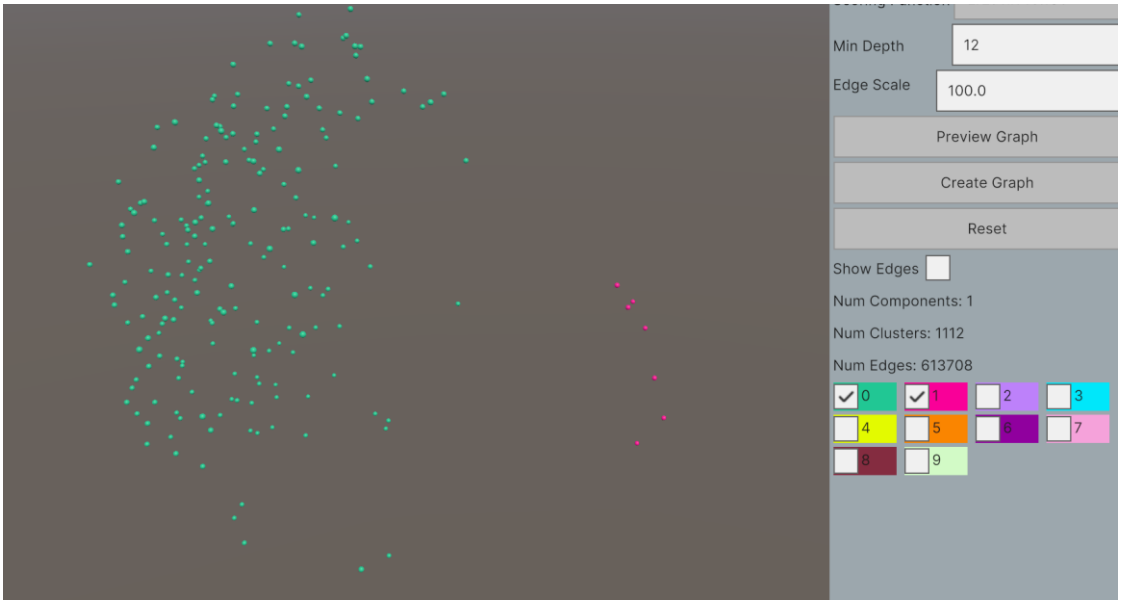


*Figure 42: 0 (green) and 1(red) are on opposite sides of the MNIST graph.*

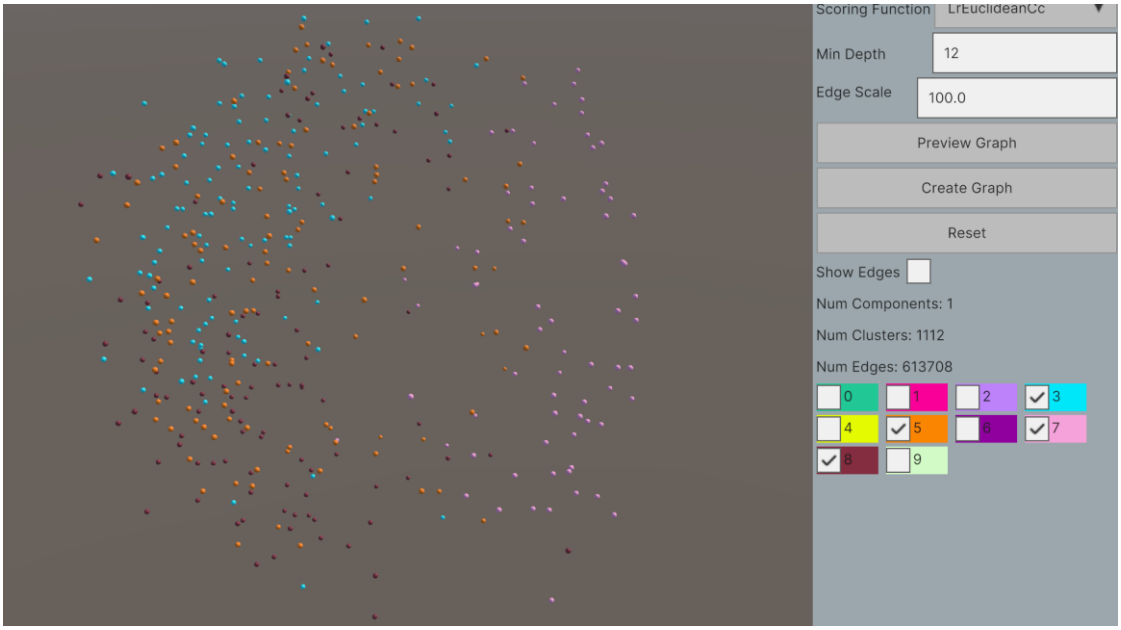


*Figure 43: The graph displayed with digits 3, 5, and 8 grouped on the left. Digit 7 is on the right for reference.*

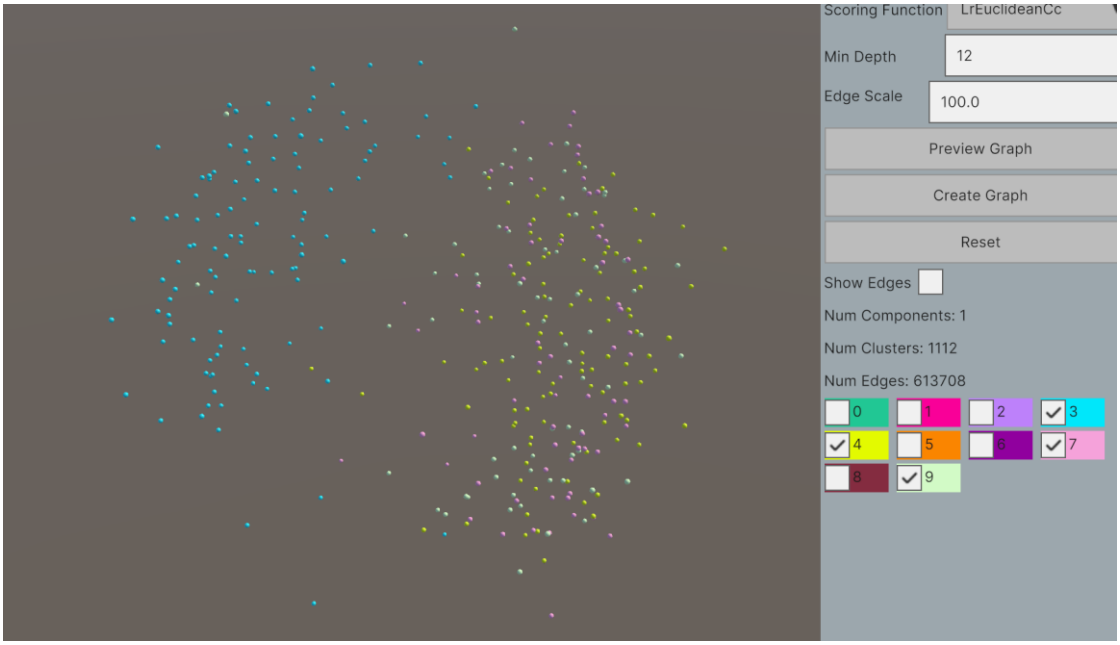


*Figure 44: Digits 4,7, and 9 are grouped on the right. Digit 3 is displayed on the left for reference.*

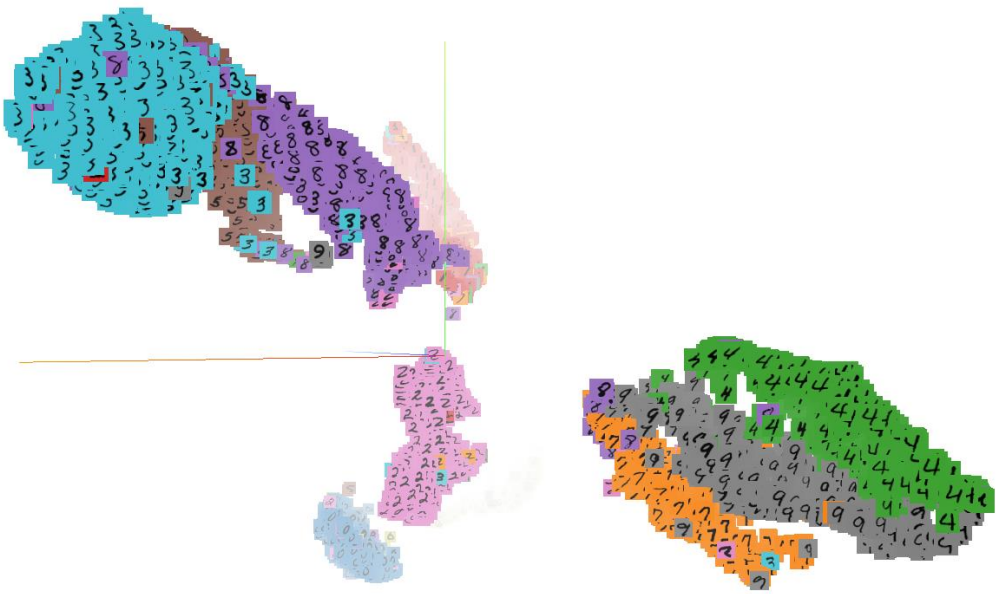

*Figure 45: A UMAP visualization of MNIST in 3D showing groupings of 3,5, 8 and 4,7, 9.*

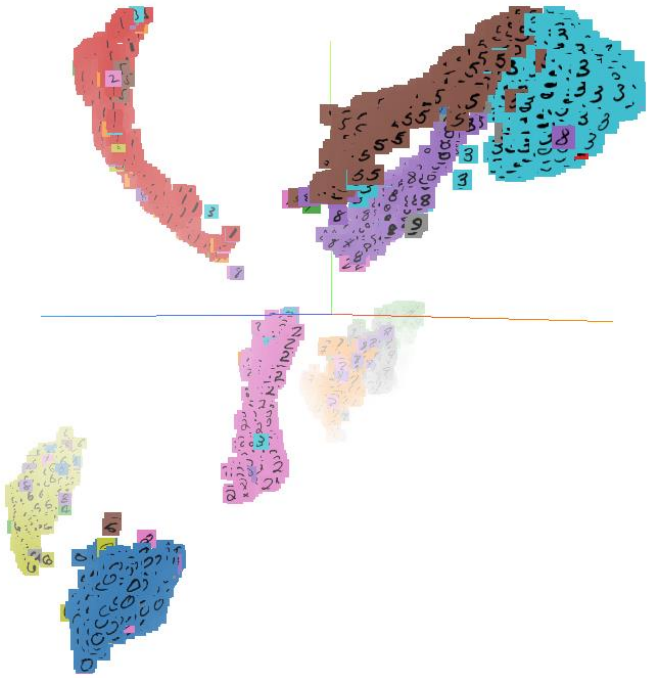

*Figure 46: A UMAP visualization of MNIST in 3D focusing on the relationship between 0 and 1 digits.*

### 4.2.2 An Analytical Comparison

Both the authors of the UMAP paper and Cayton state that it is difficult to measure the accuracy of the manifold created through manifold learning. For a quantitative analysis on the performance of my visualization, I chose to leverage the geometrical properties of triangles. To that end, I would randomly select three clusters in my graph and use their physical positions in 3D space to form a triangle. I would then find the metric distance (in my experiments I used Euclidean) between the three clusters as the reference triangle. I also ensured that the points selected made valid triangles by testing the triangle inequality theorem, which states that the sum of the lengths of any two sides of a triangle must be greater than or equal to the length of the remaining side [20]. Once this was done, I used three tests to quantify the accuracy of my graphs.

The initial evaluation involved assessing the lengths of the edges of each triangle. In this analysis, a reference triangle comprised points A, B, C, while a test triangle consisted of points A', B', C'. The primary objective was to ensure that the edges maintained the correct relative order by length. For instance, if AB > AC > BC in the reference triangle, then A'B' > A'C' > B'C' in the test triangle would indicate that the distances between the data points in 3D space and their embedding space were proportional.

I defined the accuracy of the graphs as the ratio of triangles with proportional edges to the total number of sampled triangles. However, determining what constitutes a "good" score remained subjective. To address

this, I conducted similar tests on graphs generated by UMAP and compared the accuracies. The only variation in these tests was the manipulation of the number of neighbors when constructing UMAP graphs, while I tested my graphs with a variable number of min_depth parameters as these parameters can impact the accuracy of the graphs. I chose to plot the results for UMAP and clam separately because the minimum depth and number of neighbors hyperparameters are not equivalent, i.e. a minimum depth of five does not directly compare to a number of neighbors of five. The purpose of creating graphs with a variety of hyperparameters like this is to evaluate how they can affect the accuracy of the graphs. Another difference to keep in mind is that I am comparing the accuracy of distances portrayed between clusters with the clam graph whereas the distances are between datapoints with the UMAP graphs.

The remaining two tests aimed to quantify the extent of distortion present in the triangles, rather than simply identifying whether distortion occurred. In one test, I compared the lengths of the edges, while in the other, I examined the magnitude of the angles. In the first test, the focus was on assessing the proportion of the perimeter of a triangle occupied by each edge in their respective spaces. This measurement provided insight into the extent to which edges were stretched or compressed when transitioning between dimensions. The exact equation used to measure the distortion of a triangle can be seen below.

$$refEdgeRatio = \frac{len(AB)}{perimeter(refTriangle)}$$

$$testEdgeRatio = \frac{len(A'B')}{perimeter(testTriangle)}$$

$$edgeDistortion = abs(refEdgeRatio - testEdgeRatio)$$

$$triangleDistortion = disortionAB + distortionAC + distortionBC$$

$$avgTriangleDistortion = triangleDistortion \,/\, 3.0$$

The second test measures the distortion of the angles between the points. I calculated the angles of the triangle in the original embedding space by using the law of cosines [21]. This is because they are not aligned to a coordinate system, and I needed to calculate the angles while only knowing the edge lengths of the triangle. Once I have calculated the angles, I take ratios in a similar method as described with edge distortion (except using 180 degrees instead of the perimeter). I plotted these results so that triangle_equivalence represents the percentage of triangles that had edges sorted in the same order while edge and angle accuracy are 1 – the respective distortion percent (meaning that a higher value is better for all three tests).

| Dataset | Points | Dimensionality |
|---|---|---|
| Arrhythmia | 452 | 274 |
| Mnist | 7603 | 100 |
| Satellite | 6435 | 36 |
| Wine | 129 | 13 |

*Table 2: The number of points and dimensionality of datasets used to compare accuracy of CLAM graphs and UMAP graphs.*

I ran these tests on datasets with a variety of dimensionalities seen in Table 2 above. Figure 47 and Figure 48 show accuracy of the clam and umap graphs with the Arrhythmia dataset. The results of the edge and angle accuracy tests show that UMAP and clam are relatively close, with both being above 80% accuracy. However, the clam graph does slightly outperform the UMAP graph. An interesting observation is that the sorted edges test has a significant difference, with the accuracy of the clam graph hovering between 65% and 85% accuracy while the UMAP graph is between 30% and 40$ accuracy. This shows the importance of measuring not just if the edges were distorted, but by how much. For example, this infers that roughly 70% of the UMAP triangles did not have edge lengths in the same order as the original embedding space, however the average distortion of the edges was about 20%, meaning that they were still relatively accurate.

In Figure 49 and Figure 50, I compare the accuracy of the graphs created with the MNIST dataset. Building a graph with clam from a minimum depth of 4 results in a lower accuracy than graphs created with UMAP, however at all other depths clam outperforms UMAP in accuracy. However, the performance in the edge sorting test is much closer than with the Arrhythmia dataset. Figure 51 and Figure 52 show that clam outperforms UMAP in all three tests with the Satellite dataset.

Figure 53 and Figure 54 compare the results of graphs built with the Wine dataset. The results plotted here are interesting because they show that clam graphs have an accuracy of about 100% at depths four through eight.

However, depths between nine and eleven show a sharp drop off in accuracy because its tree has a height of eleven. Leaf Clusters of a tree with a cardinality of one have a radius of zero, so they are less likely to form edges with other clusters, especially other leaf clusters. If a graph consists entirely of leaf clusters, it ends up with no edges, which essentially means the local structure of graph is lost.

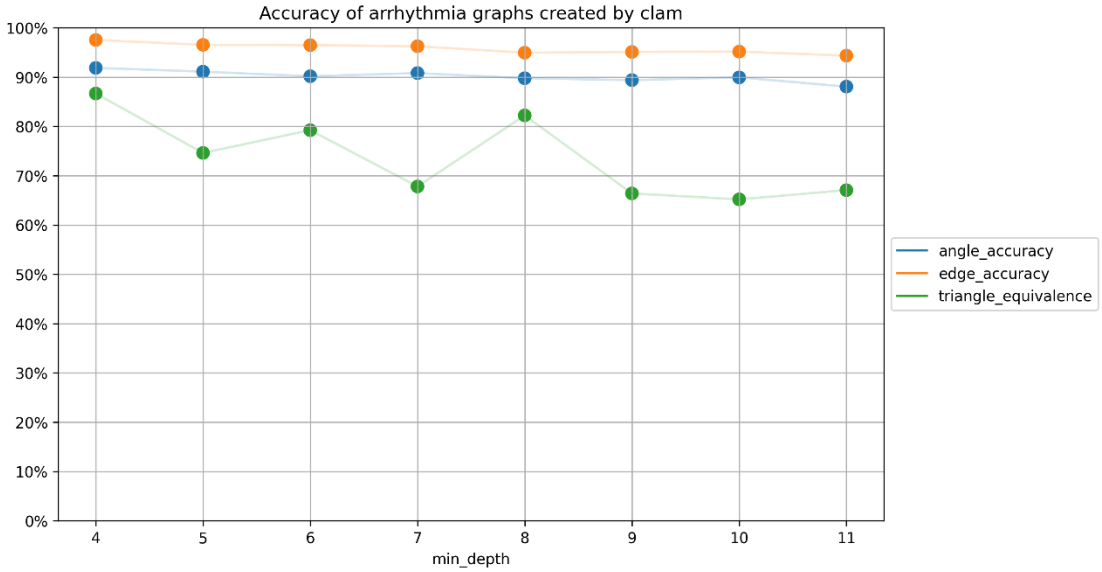


*Figure 47: Arrhythmia accuracy at various depths as visualized by CLAM.*

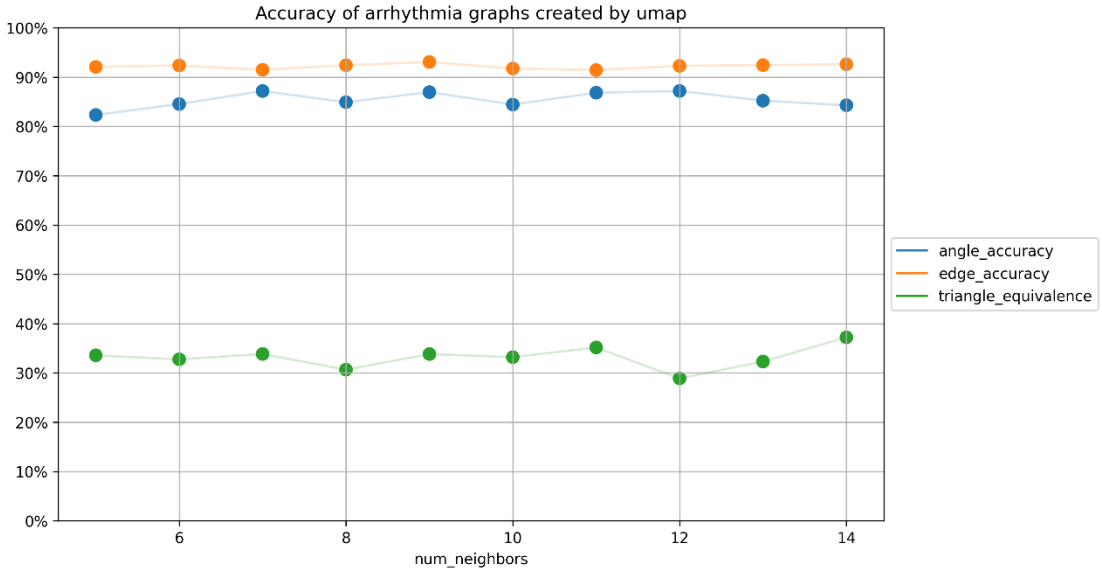


*Figure 48: Accuracy of the Arrhythmia dataset as portrayed by UMAP.*

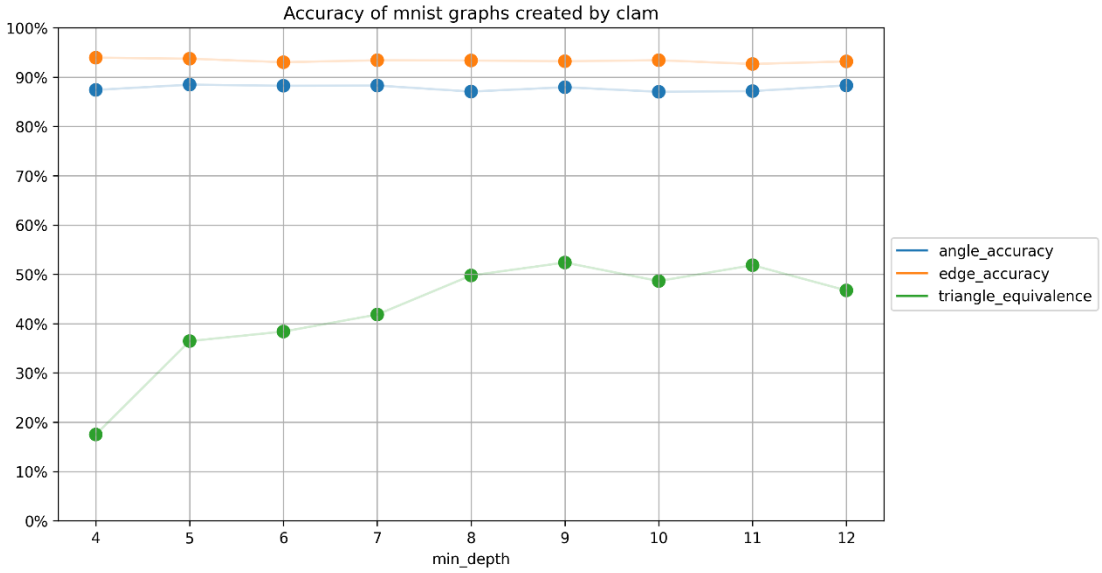


*Figure 49: The accuracy of the MNIST dataset graph visualized by CLAM.*

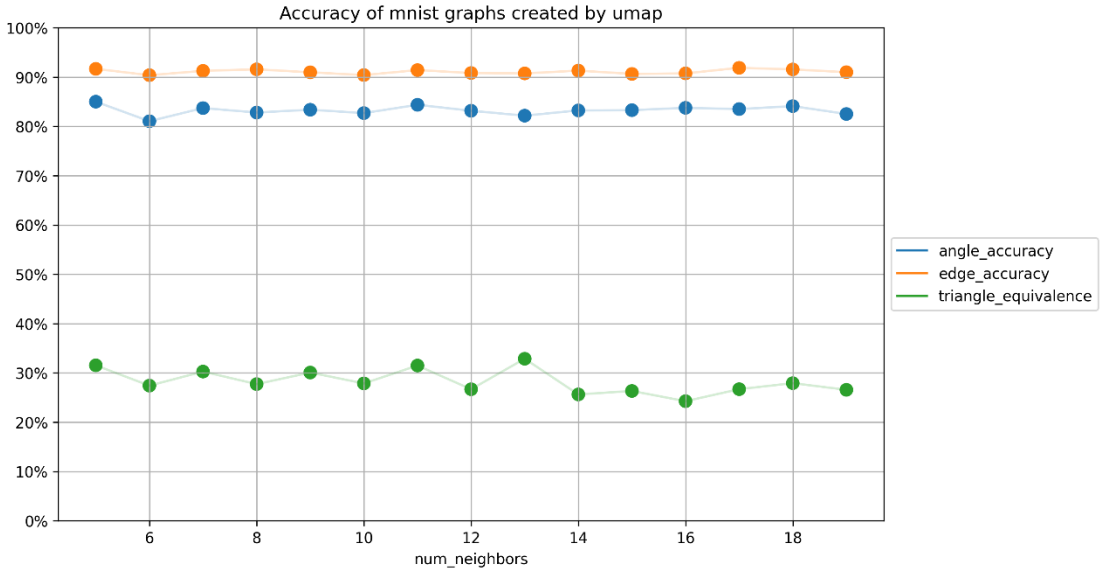


*Figure 50: Accuracy of the MNIST dataset as portrayed by UMAP.*

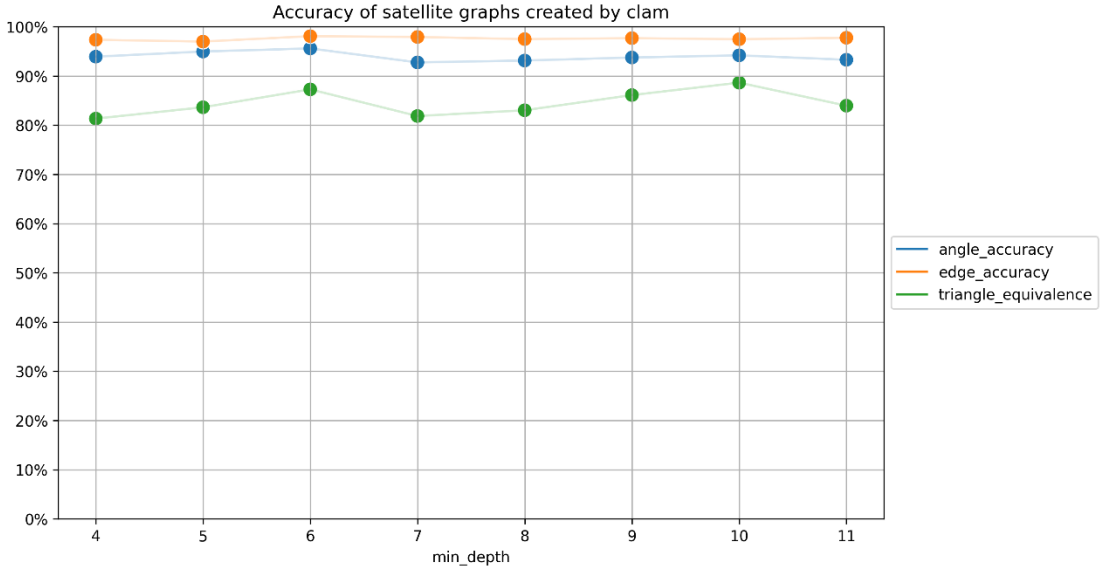


*Figure 51: The accuracy of the Satellite dataset portrayed by CLAM.*

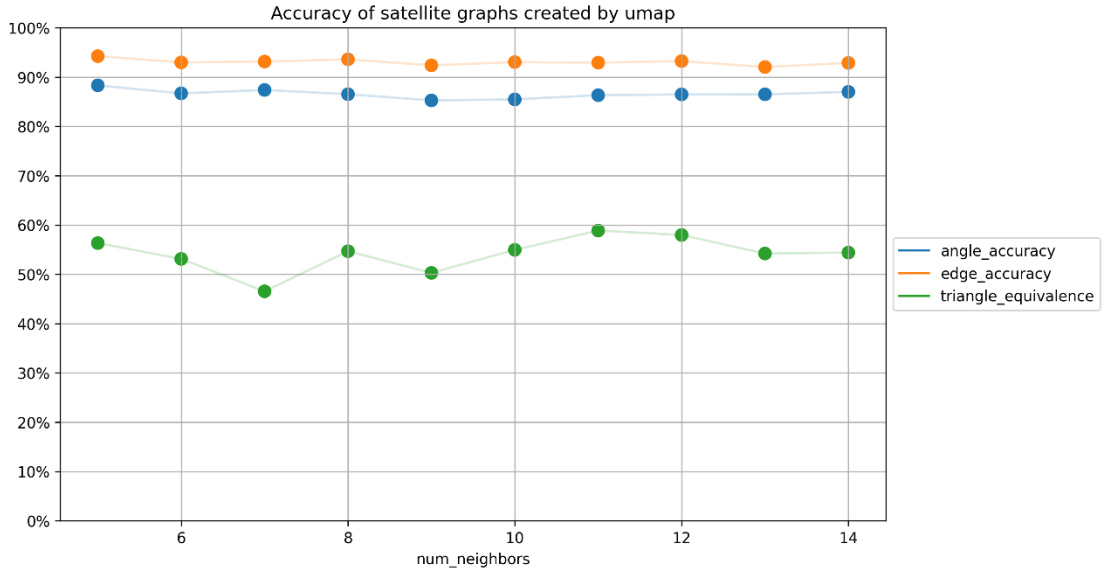


*Figure 52: Accuracy of the Satellite dataset as portrayed by UMAP.*

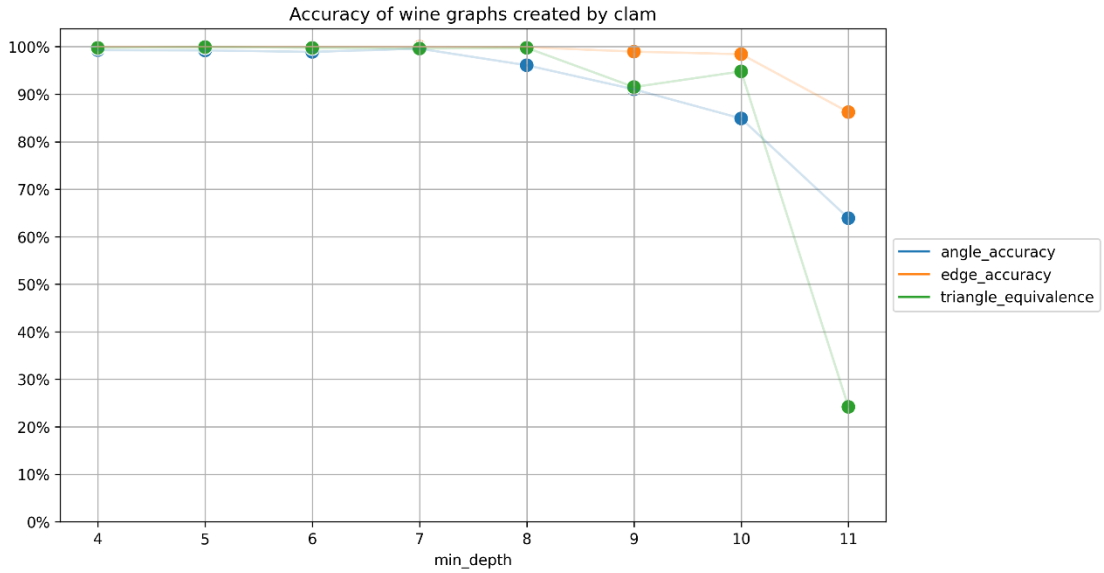


*Figure 53: Accuracy of the Wine dataset portrayed by CLAM at various depths.*

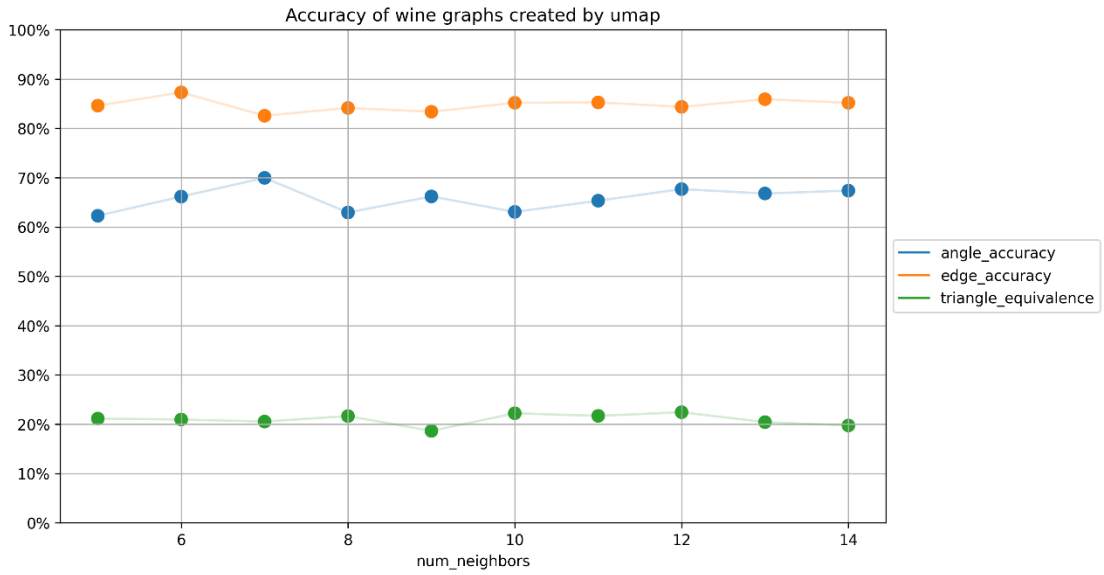


*Figure 54: Accuracy of the Wine dataset as portrayed by UMAP.*

One other test I ran was to see how much the accuracy of the graphs improved over the course of the physics simulation. The plots below show the accuracy of the graphs during each time step. The edge and angle accuracies are not inverted in the graphs below, so a lower score is better for those two graphs. The results show that the physics forces are improving the quality of the graph over time by decreasing distortion and increasing the number of edges that are sorted in the same order between embedding spaces.

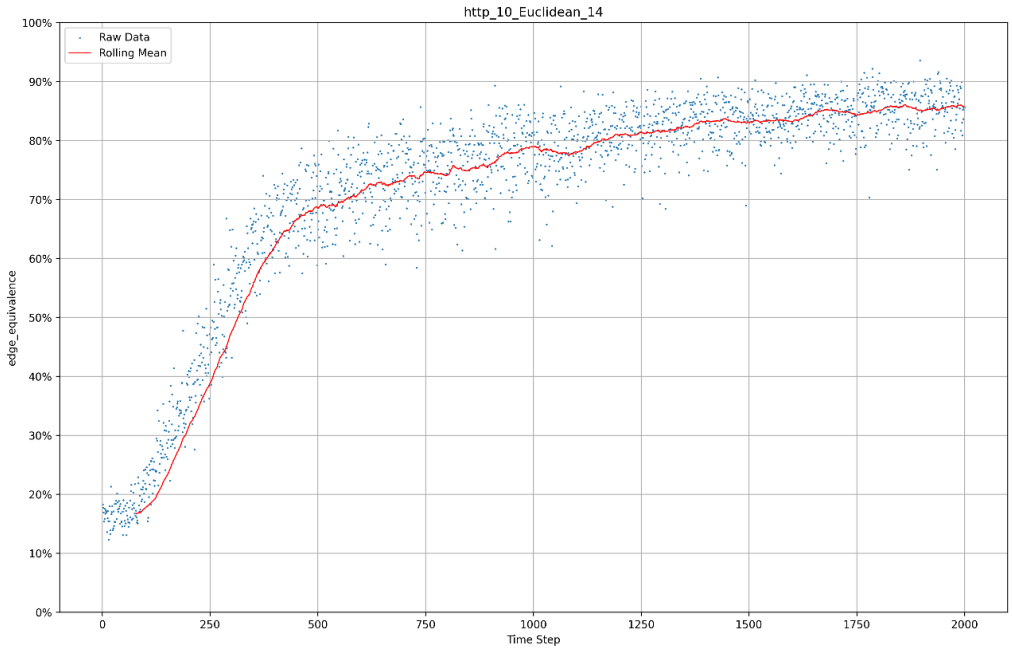


*Figure 55: The percentage of triangles that are proportional in 3D and original embedding space during each step of the physics simulation.*

Quantifying the accuracy of the graph representation of the manifold can be a difficult task. By creating these tests, I am showing that the graphs induced with CLAM provide an accurate visualization of the structure of the underlying manifold. My visualization put more of an emphasis on the importance of preserving the distances between vertices in the graph than methods such as UMAP and t-SNE. This can be reflected by the fact that the clam graphs are more accurate than the UMAP graphs when comparing their performance on several datasets with a variety of dimensionalities.

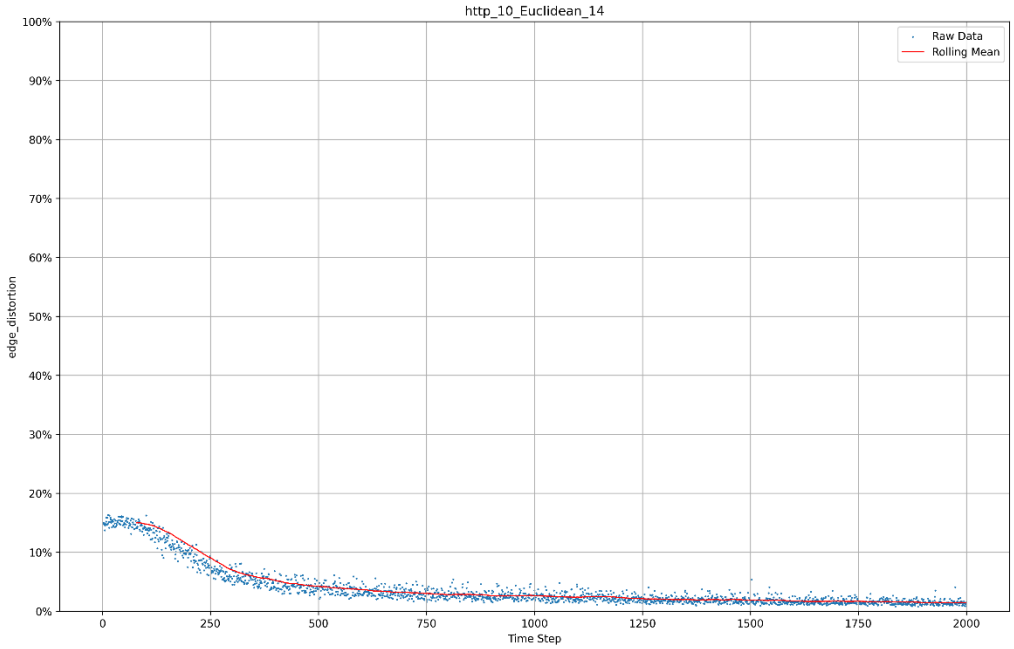


*Figure 56: The average distortion of the edges in the http graph during each time step of the physics simulation.*

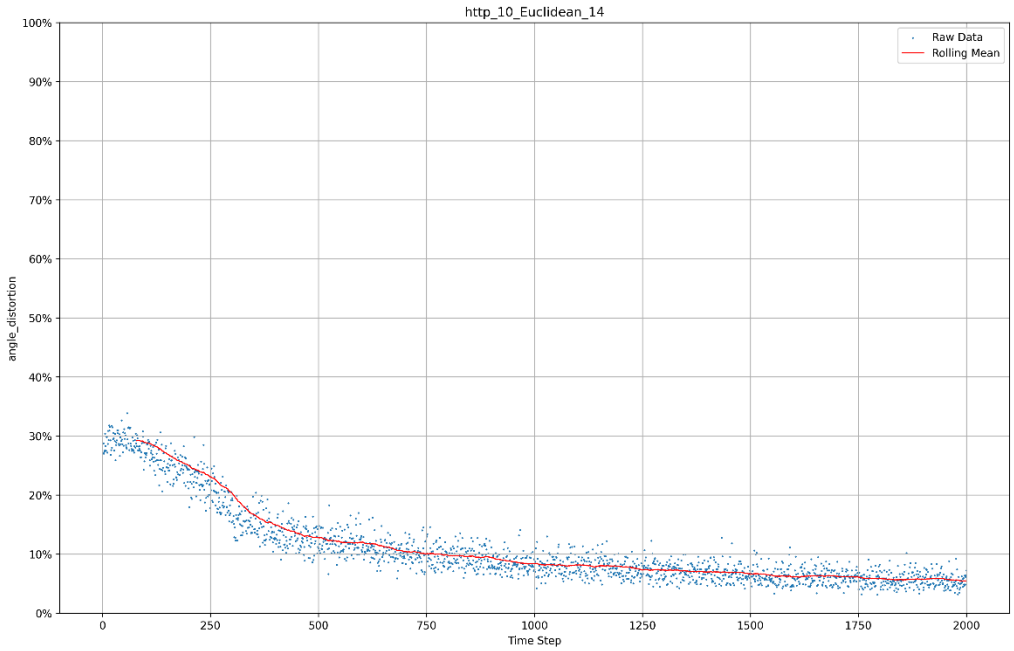


*Figure 57: The average distortion of the angles in the http graph during each time step of the physics simulation.*

CHAPTER 5

# CONCLUSION

In conclusion, the application of clustered manifold mapping as a novel visualization technique has demonstrated significant potential in addressing the challenges of visualizing high-dimensional data. Visualizing the tree that CLAM creates provides an interesting summary of the dataset as well as providing additional insights into the field of clustered manifold mapping. Additionally, the intermediate step of highlighting clusters in the tree that have been selected for building the graph provides the user with an intuitive understanding of the “resolution” at which they will be viewing the manifold. Moreover, the resulting graph visualization enables users to delve into complex datasets, revealing the relationships between closely associated or disparate clusters of data.

The qualitative and quantitative analysis of the performance of this visualization technique in comparison to existing methods such as UMAP demonstrates its effectiveness in showing distances between data points. This means that it not only groups similar data points well but also accurately represent their distance in the original data space and produces an accurate visual representation of the underlying structure of the manifold.

FUTURE WORK

This thesis not only provides an important contribution to the field of visualizing high dimensional data, but it also paves the way for future work in the field of clustered manifold mapping. Examples of features that could provide valuable insight to these complex datasets would be coloring clusters by graph component, allowing the user to show or hide cluster and edges based on certain variables, and allowing the user to dynamically change the resolution of the graph by replacing a cluster with its children (or vice versa).

Human computer interaction was not the primary focus of my thesis, however future work could improve the user's experience by streamlining various parts of the user interface. In a small HCI experiment I conducted, some feedback I received included comments on the "clunky" nature of switching between the overlayed user interface and the in-world UI (i.e. clicking clusters and moving the camera). Other comments were made on how the use of a dropdown menu slowed down users because they might need to switch menus to complete a task. Towards the end of my work as part of a course I was taking, I created a rough draft of a more user-friendly UI based on this feedback, which can be seen in figure 38. Additionally, there is potential to create a VR version of this application that would allow the user to interact with the data with their hands rather than a keyboard and mouse.

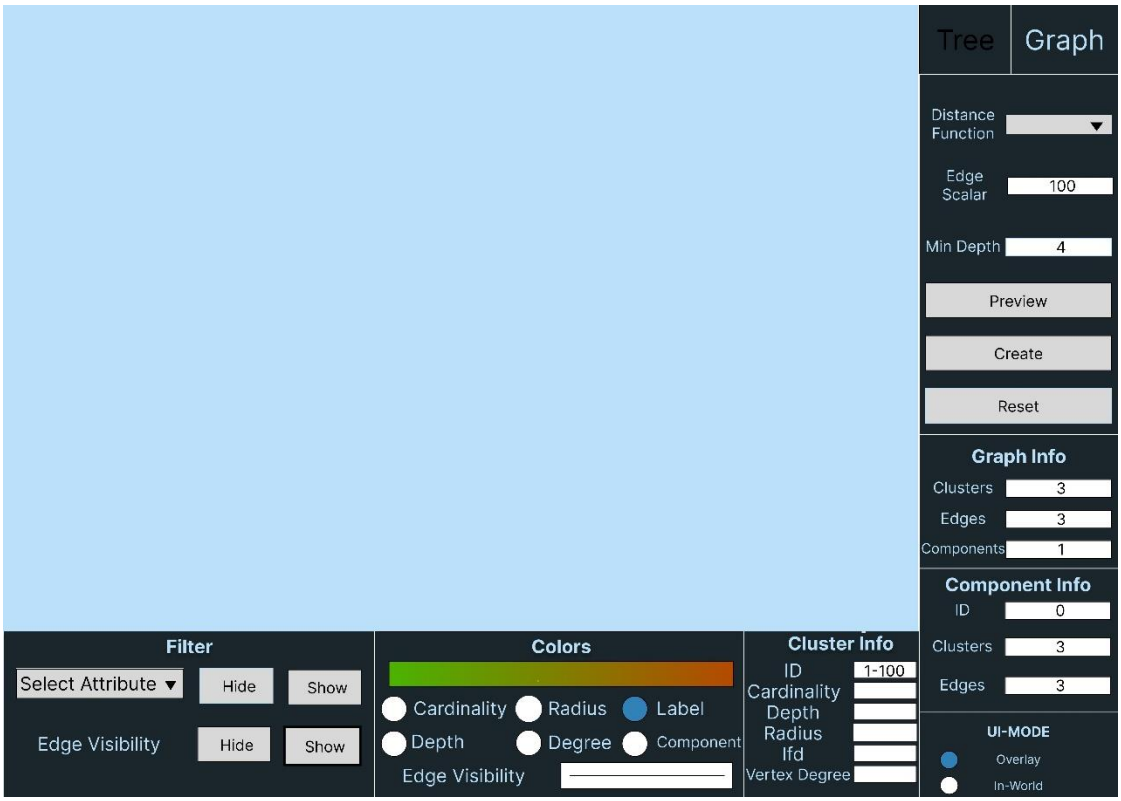


*Figure 58: An experimental UI design that could improve the human computer interaction of the application.*

Another aspect of potential improvement for this work could be on the performance of the graph building and physics simulation. As I noted in the methodology section, despite calculating the physics forces on a worker thread, there can still be noticeable performance hits on larger graphs because of the sheer number of clusters that need to be processed each frame. Looking into the use of multithreading to perform the calculations and updates of cluster positions in parallel could prove quite useful to improving the performance of the application.

In the methodology section, I noted that the idea of applying global forces between disjoint components without disrupting the local geometry seems

promising and laid out an algorithm for doing so that should be studied and implemented in more depth.

Another area of future work could be further developing the tests to show how accurate the graphs are. One such test being discussed in my research group is “false nearest neighbors.” This test would find the k-nearest neighbors of each cluster in the graph and compare it with the k-nearest neighbors of the datapoints in their original embedding space. If the same neighbors are found, that would indicate the graphs do a good job of finding the k-nearest neighbors. If the same neighbors are not found, it would indicate that the graph does a poor job of representing which clusters are most closely related to other clusters.

# APPENDICES

*Link to Source Code: https://github.com/djperrone/clam_visual3d*

BIBLIOGRAPHY

[1] D. Probst and J.-L. Reymond, “Visualization of very large high-dimensional data sets as minimum spanning trees,” *Journal of Cheminformatics*, vol. 12, no. 1, Feb. 2020, doi: https://doi.org/10.1186/s13321-020-0416-x.

[2] L. Cayton, “Algorithms for manifold learning,” UC San Diego: Department of Computer Science & Engineering, 2005. Accessed: Apr. 14, 2024. [Online]. Available: https://www.cs.columbia.edu/~verma/classes/ml/ref/lec8_cayton_manifolds.pdf

[3] H. S. Seung, “COGNITION: The Manifold Ways of Perception,” *Science*, vol. 290, no. 5500, pp. 2268–2269, Dec. 2000, doi: https://doi.org/10.1126/science.290.5500.2268.

[4] L. McInnes, J. Healy, and J. Melville, “UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction,” *arXiv.org*, 2018. https://arxiv.org/abs/1802.03426

[5] N. Ishaq, T. J. Howard III, and N. M. Daniels, “Clustered Hierarchical Anomaly and Outlier Detection Algorithms,” *arXiv.org*, Nov. 21, 2021. https://arxiv.org/abs/2103.11774 (accessed Apr. 14, 2024).

[6] D. Donoho, “Aide-Memoire. High-Dimensional Data Analysis: The Curses and Blessings of Dimensionality,” 2000. Accessed: Apr. 14, 2024. [Online]. Available: https://www.math.ucdavis.edu/~strohmer/courses/270/Donoho_Curses.pdf

[7] W. W. B. Goh and L. Wong, “The Birth of Bio-data Science: Trends, Expectations, and Applications,” *Genomics, Proteomics & Bioinformatics*, vol. 18, no. 1, pp. 5–15, Feb. 2020, doi: https://doi.org/10.1016/j.gpb.2020.01.002.

[8] A. N. Gorban and I. Y. Tyukin, “Blessing of dimensionality: mathematical foundations of the statistical physics of data,” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 376, no. 2118, p. 20170237, Mar. 2018, doi: https://doi.org/10.1098/rsta.2017.0237.

[9] C. Fefferman, S. Mitter, and H. Narayanan, “Testing the manifold

hypothesis," *Journal of the American Mathematical Society*, vol. 29, no. 4, pp. 983–1049, 2016, Accessed: Apr. 14, 2024. [Online]. Available: https://www.jstor.org/stable/jamermathsoci.29.4.983
[10] S.-H. Cheong, Y.-W. Si, and R. K. Wong, "Online force-directed algorithms for visualization of dynamic graphs," *Information Sciences*, vol. 556, pp. 223–255, May 2021, doi: https://doi.org/10.1016/j.ins.2020.12.069.
[11] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and Experience*, vol. 21, no. 11, pp. 1129–1164, Nov. 1991, doi: https://doi.org/10.1002/spe.4380211102.
[12] C. John and Holton Derek Allan, *A First Look at Graph Theory*. Allied Publishers, 1995, p. 28.
[13] I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150202, Apr. 2016, doi: https://doi.org/10.1098/rsta.2015.0202.
[14] J. Lever, M. Krzywinski, and N. Altman, "Principal component analysis," *Nature Methods*, vol. 14, no. 7, pp. 641–642, Jul. 2017, doi: https://doi.org/10.1038/nmeth.4346.
[15] L. van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008, Available: https://jmlr.org/papers/v9/vandermaaten08a.html
[16] M. Prior, T. Iii, O. Mclaughlin, T. Ferguson, N. Ishaq, and N. Daniels, "LET THEM HAVE CAKES: A CUTTING-EDGE ALGORITHM FOR SCALABLE, EFFICIENT, AND EXACT SEARCH ON BIG DATA." Accessed: Apr. 14, 2024. [Online]. Available: https://arxiv.org/pdf/2309.05491.pdf
[17] J. Yallop, D. Sheets, and A. Madhavapeddy, "A modular foreign function interface," *Science of Computer Programming*, vol. 164, pp. 82–97, Oct. 2018, doi: https://doi.org/10.1016/j.scico.2017.04.002.
[18] M. Schoonmaker, "Getting started with FFI: Rust & Unity," *Test Double*, Jan. 02, 2018. https://blog.testdouble.com/posts/2018-01-02-unity-rust-ffi-getting-started/ (accessed Apr. 14, 2024).

[19] E. M. Reingold and J. S. Tilford, “Tidier Drawings of Trees,” *IEEE Transactions on Software Engineering*, vol. SE-7, no. 2, pp. 223–228, Mar. 1981, doi: https://doi.org/10.1109/tse.1981.234519.
[20] M. A. Khamsi and W. A. Kirk, *An Introduction to Metric Spaces and Fixed Point Theory*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2001. doi: https://doi.org/10.1002/9781118033074.
[21] Clarence Raymond Wylie, *Plane Trigonometry*. McGraw-Hill, 1955.