MILO RAZIEL SANTOS RODRIGUES

ANALYZING THREE-DIMENSIONAL GRAPH-DRAWING HEURISTICS ON PROTEIN INTERACTION DATA AND GENERAL GRAPHS

This dissertation was presented to the Department of Computer Science at Universidade Federal da Bahia (UFBA), as a requirement to obtain a Bachelor of Science degree in Computer Science.

Adviser: Ricardo Araújo Rios

Salvador
December 20th, 2022

ACKNOWLEDGEMENTS

I would like to thank my adviser, Prof. Ricardo Araújo Rios, for sticking with this project until the end. My mother, for inspiring me to follow her into Computer Science in the first place, and being willing as only a mother would to field all of my endless calls as this project progressed. My friends, for listening to my monologues as I scribbled vectors and formulas on the edges of napkins. Without their guidance and support this project would not have been possible.

As this project marks a very significant moment in my life, I would also like to thank all my professors and supervisors over these last few years, not only for providing me with knowledge but also the tools and drive to seek it on my own. To all the people in my life who were willing to stick with me through the endless highs and lows, I can never put into words just how grateful I am to you.

Thank you.

The choice to live is not made just the once, after all.

—JAMMINGWAY (FINAL FANTASY XIV)

RESUMO

Grafos são uma estrutura de dados comumente utilizada em diversas áreas da Ciência da Computação, buscando-se aproveitar sua flexibilidade e sua capacidade de modelar informação relacional de maneira robusta e elegante. A utilização de dados em formato de grafos é comum em múltiplas áreas de pesquisa, incluindo estudos sobre redes sociais, interação de proteínas e design de substâncias, design de circuitos, etc. Nos últimos anos, avanços na área de *deep learning* têm exibido bons resultados no desenvolvimento de métodos de aprendizado de máquina orientados a grafos. Porém, um dos problemas comumente encontradas no trabalho com grafos é a dificuldade de representar a informação contida no grafo (que abrange não só atributos, mas também estruturas) de uma maneira humanamente interpretável. Uma das abordagens a esse problema consiste em técnicas de representação gráfica de grafos. Este trabalho busca apresentar uma prova de conceito de uma ferramenta de desenho tridimensional de grafos escrita em Python e OpenGL, visando uso geral mas desenvolvida para utilização de pesquisadores da área de aprendizado de máquina com grafos, em particular estudo de interação de proteínas. Neste trabalho, investigamos e implementamos alguns dos algoritmos mais conhecidos de desenho de grafos, incluindo adaptações para situações específicas, e analisamos sua performance em grafos generalizados e em dados reais utilizados em aprendizado de máquina, considerando-se grafos grandes e pequenos.

Palavras-chave: desenho de grafos, grafos tridimensionais, interação de proteínas, algoritmos direcionados por forças

ABSTRACT

Many fields of computer science favor the usage of graph-modeled data, such as research into social networks, protein interaction and circuit design. Recent advancements in deep learning have shown promising results from graph-oriented machine learning, as the relational information described by a graph can be a powerful asset in performing predictions. One common challenge in working with graphs lies in presenting the information contained therein in a way that is suitable for human interpretation, which is the driving motivation behind the field of graph drawing. This work presents a proof of concept for a three-dimensional graph-drawing tool written in Python and OpenGL, aimed at research into machine learning models utilizing graph data, specifically protein interaction. We review and implement some of the better known extant force-directed graph-drawing algorithms, analyzing their performance on both general graphs and real-life data utilized in machine learning, for both small and large graphs.

Keywords: graph drawing, protein interaction, force-directed, three-dimensional graphs

CONTENTS

LIST OF ALGORITHMS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Graphs are abstract data structures generally used to model domain-significant relationships between instances in a set of data. In a given graph $G(V, E)$, V is a set of entities represented as *nodes* or *vertices*, while the set E represents the relation as the *edge* (u, v) belongs to E if the nodes u and v are related (DI BATTISTA et al., 1999). The versatility and flexibility of a graph allow robust modeling of many types of data, making it one of the most conventional structures to represent relational information.

Like in most areas of computational science, graphs are also a useful tool in the field of artificial intelligence, where their expressive power is a great asset in modeling a variety of network-based systems, from social to natural science. Literature on the application of machine learning methods on data modeled as graphs dates back to the 1990s, and recent advancements in deep and convolutional neural networks have brought renewed attention to the field, from the study of GNNs (Graph Neural Networks) to research into graph representation learning (ZHOU et al., 2020). The modeling of data as graphs allows a neural network to better grasp the inductive bias present in the data, in the complex relations and element interactions a graph models, by virtue of being a network. Usage of graphs in machine learning, especially GNNs, has yielded robust results in applications such as community detection, recommendation systems, molecular analysis (including medicine design), among others (WU et al., 2022).

However, one main issue in dealing with graphs is that much of the relational information they hold is not always easily discernible to the human eye, meaning significant parts of it may go unnoticed in analysis. This limitation creates the necessity for graph representations, or embeddings, that maximize human readability, and a demand for efficient, automatic production of such representations. Two-dimensional techniques have been widely studied for over six decades (TUTTE, 1963), and there is growing interest and research into the development of styles and algorithms for three-dimensional drawings. Certain modern applications and emerging technologies also benefit from 3D graph display over 2D and drive the demand for such research, from VLSI (Very Large Scale Integration, the process of creating an integrated circuit that contains millions to billions of transistors in a single chip) circuit design to information visualization to nanotechnology (DUJMOVIC; WHITESIDES, 2013).

When it comes to three-dimensional graph drawing, there has been a good chunk of research into the category of *force-directed* drawing algorithms, also known as spring embedders, a class of algorithm that models a graph as a physical object in which its nodes exert forces of attraction and repulsion on one another. These modeled forces are then allowed to act, aiming to naturally achieve a state of static equilibrium (FRUCHTERMAN; REINGOLD, 1991) as a physical system would. The original spring embedder designed by Peter Eades has served as a starting point (EADES, 1984), modeling the forces with similar behavior as their real-life references. Researchers have since tackled further questions, such as the performance of different metrics in the modeling like the usage of shortest-path distances rather than Euclidean (KAMADA; KAWAI, 1989) and method improvements to accommodate large graphs (HADANY; HAREL, 2001).

While academic research on the topic has advanced, there is still a shortage of general-purpose visualization tools for use in graph-related research, especially in the field of artificial intelligence and, particularly, machine learning, to enable more straightforward human interpretation of both data and results. The research group this work is geared towards is currently studying severity prediction in hemophilia patients, modeling the defective proteins that cause the disorder as graphs (LOPES et al., 2021b), on which machine learning models can then be utilized (LOPES; NOGUEIRA; RIOS, 2022). In this case, a graph visualization tool could be greatly beneficial in more easily observing relationships between the amino acids.

This work aims to present a proof of concept for one such tool, employing force-directed drawing in a three-directional space to offer flexible, human-friendly visualization, and implement and analyze the efficacy of extant graph drawing algorithms on graph data, both general and related to the research above.

This dissertation is organized as follows: in the next section, we discuss force-directed models for drawing graphs; then, we present our contribution; next, we show the experimental setup and results; and, finally, conclusions are drawn in the last chapter.

Chapter 2

FORCE-DIRECTED GRAPH DRAWING MODELS

2.1 OVERVIEW

The goal of any given graph drawing algorithm is to automatically produce an embedding that maximizes a set of certain aesthetic criteria, often referred to as "readability" (DI BATTISTA et al., 1999) to encompass the degree of ease of human interpretation of said embedding, but sometimes also described in more aesthetic-related terms such as "beauty" or "niceness" (HAREL; KOREN, 2000). Such criteria generally include requirements such as avoidance of tight clusters and minimization of edge crossings, but further criteria may be required on a domain by domain basis, such as the shape of the edges (e.g. most three-dimensional applications favor straight-line drawings, while graphs that represent circuits are often drawn with orthogonal, grid-like edges), the display of symmetries, planarity, drawing size, etc.

With that in mind, the problem of graph drawing can then be approached as an optimization problem, modeling the metric of "readability" as a function that can be maximized (or, conversely, minimize *un*readability). In the scope of three-dimensional graph drawing, one of the leading approach has been through the use of *force-directed algorithms* (DUJMOVIC; WHITESIDES, 2013), due to their flexibility, ease of use, and intuitive synergy with the three-dimensional medium. These algorithms abstract the influence of nodes' positions on one another as imaginary physical forces, exerted on each node by every other node in the graph, which then makes it possible to compute a (usually local) minimum energy configuration (DI BATTISTA et al., 1999), which is then expected to produce an aesthetically pleasing drawing (GAJER; GOODRICH; KOBOUROV, 2000). In this approach, only information contained within the graph is used, regardless of domain-specific knowledge (KOBOUROV, 2013), producing embeddings that emphasize the graph's actual structure and minimize interpretation bias.

This abstraction describes each node v as being under the influence of "forces" exerted by other nodes, often done by modeling each edge as a spring - that is, exerting attractive force inversely proportional to its current length (DI BATTISTA et al., 1999). Depending on the implementation, these forces might be gravity-like (i.e. a node can only exert force on its neighbors, and all forces are attractive), electromagnetism-like (i.e.

a node is attracted by its neighbors and repelled by non-neighbors), or any such model. Implementations might also include the same formulas and behavior as their inspiring forces present in nature, or alter them. In general, the goal is to keep a node closer to its neighbors than it is to non-neighbors.

Basic force-directed algorithms are better suited for smaller graphs, mainly due to the fact that the function that calculates the total of forces acting inside a graph has many local minima, and the algorithms struggle to consistently output good layouts even with the use of heuristics to avoid said local minima (KOBOUROV, 2013). In more recent decades, ideas to improve the scalability of force-directed drawing have commonly included multi-level approaches going from fine to coarse adjustments, laying out the simplest structures first and then tackling larger structures as they grow progressively more complex (HADANY; HAREL, 2001).

2.2 BARYCENTRIC ALGORITHM

One of the pioneers of its kind is William T. Tutte's barycentric two-dimensional drawing algorithm (TUTTE, 1963), later retroactively considered a force-directed algorithm by Di Battista *et al* (DI BATTISTA et al., 1999). The main goal of the barycentric algorithm is to place each node v as close as possible to the barycenter of the polygon formed by v's direct neighbors. After enough iterations, the algorithm converges into a more human-readable two-dimensional embedding that is aesthetically pleasing.

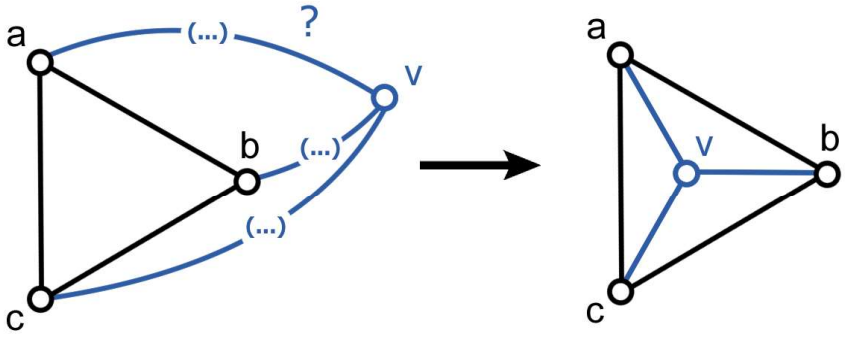

Figure 2.1 Example of barycentric placement of a node v based on its neighbors.

Given a graph $G = (V, E)$, each iteration of the algorithm positions each node $v \in V$ in the barycenter of its neighbors until convergence is achieved. Therefore, for each $v \in V$, with $N(v)$ referring to v's direct neighborhood, a given iteration can be described as:

$$(x_v, y_v) = barycenter(N(v)) = \sum_{(u,v)\in E} (x_u, y_u)/degree(v) \tag{2.1}$$

The goal can be represented by a system of linear equations: essentially, one system per dimension, all structured in the same way. The systems can then be solved. However, since there are infinite trivial solutions consisting in simply placing all nodes at the same

point (KOBOUROV, 2013), it is necessary for the user to pre-select a set of *fixed nodes*, whose position is not altered by the program and provide to the other nodes, dubbed *free nodes*, a set of spatial anchors. This way, removing the equations pertaining to the fixed nodes, the system now has a single solution (TUTTE, 1963).

Knowing that every 3-connected graph has a unique planar embedding (WHITNEY, 1932), the barycentric algorithm is capable of taking a known strictly convex face P of a 3-connected graph and draw the planar (crossing-free) embedding for the whole graph (TUTTE, 1963), which meets the need for a clean, human-readable representation of the graph.

Algorithm 1: Tutte's barycentric algorithm

Input: Graph G=(V,E), partitioned into $V = V_F \cup V_L$ with V_F being a set of at least three fixed nodes and V_L being a set of free nodes, a $|V_F|$-sided polygon P that represents a known strictly convex face of the planar embedding, K iterations.

place the nodes in V_F according to P;
place the nodes in V_L randomly or at the origin;
for $t = 0$ **to** K **do**
    foreach *free node* v *in* V_L **do**
        $$(x_v, y_v) = \frac{1}{degree(v)} \sum_{(u,v)\in E} (x_u, y_u)$$
    end
end

To fit into the force-directed abstraction, each edge may be thought of as a spring with a graph-homogeneous Hooke's spring constant, that is, all edges are equally "rigid", therefore equilibrium can be achieved on a node by placing it at the barycenter of the polygon formed by its neighbors, given that said polygon is strictly convex. This algorithm does not model any repulsive forces.

However, the barycentric algorithm has many limitations, especially for use in modern applications. The drawing's readability is inversely proportional to the number of nodes, given that the edges get progressively shorter the closer they are to the center of the drawing (KOBOUROV, 2013), meaning the model does not scale well. An extra parameter can be included to force minimum edge lengths, but it causes the opposite problem to the previous one, in that the full drawing can become overly large - with exponential area, in the worst case (EADES; GARVAN, 1995). Another limitation is the need to pre-select a set of nodes that are known to make up a face, an information that is hard to extract from a graph whose planar embedding is not already known.

2.3 SPRING-ELECTROMAGNETIC MODELING

The abstraction presented by Eades deals with both attractive and repulsive forces, presenting the former through modeling each edge as an spring and the latter as the elec-

tromagnetic repulsion between a pair of equally charged particles (EADES, 1984). The algorithm considers that only non-connected nodes repel each other, and only connected nodes attract each other (through the edge turned spring).

The attraction forces follow Hooke's law, in which the magnitude of the exerted force is directly proportional to the Euclidean distance between the nodes (i.e. the length of the spring), dependent on a spring rigidity constant c_1, and zero when the nodes lie at a distance of c_2 (i.e. the length of the spring when relaxed) (DI BATTISTA et al., 1999). The repulsion forces loosely model Coulomb's law on the electrostatic force between two stationary charged particles, in which they are inverse-square to the Euclidean distance between a given pair of nodes and dependant on a constant c_3 that is reminiscent of Coulomb's constant (KOBOUROV, 2013).

Given a graph $G = (V, E)$, each iteration of the algorithm calculates the resulting force exerted over each node $v \in V$ according to the rules above, assuming that every other node is, at that moment, fixed. It then applies said force to the node's current position.

Algorithm 2: Eades's spring-electromagnetic algorithm

Input: Graph G=(V,E), constants [c_1, c_1, c_3, c_4], K iterations.

place all nodes in V randomly;
for $t = 0$ **to** K **do**
  foreach v *in* V **do**
    foreach $u \neq v$ *in* V **do**
      $distance$ = Euclidean distance between u and v;
      if $(u, v) \in E$ **then**

$$\overrightarrow{FAttr_{uv}} = c_1 * log(distance/c_2)$$

      end
      else

$$\overrightarrow{FRep_{uv}} = c_3/distance^2$$

      end
    end

$$\overrightarrow{F_v} = \sum_{(u,v)\in E} \overrightarrow{FAttr_{uv}} + \sum_{(w,v)\notin E} \overrightarrow{FRep_{wv}}$$

$$v = v + (c_4 * \overrightarrow{F_v})$$

  end
end

In this implementation, the constant c_1 plays the role of the modeled spring's rigidity constant, and the logarithmic function is used to calculate the attractive forces due to the original, linear form of Hooke's law being deemed overly strong for this model (EADES,

1984). That said, the readability gain is debatable when compared to the extra computational effort (DI BATTISTA et al., 1999), so any desired function may be used as long as the property that no force is exerted when $distance = c_2$ is kept. c_3, as mentioned, plays a similar role to Coulomb's constant to regulate the strength of the repulsive forces. c_4 is an extra constant introduced to calibrate the granularity of each iteration's adjustments.

The original work targets small graphs with up to 30 nodes, in which most graphs are said to achieve a (locally) minimal energy state in up to 100 iterations (i.e. $K = 100$). The algorithm may still be applied to larger graphs, but it will require more iterations and be subject to issues with the local minima as mentioned before.

2.3.1 Kamada and Kawai's graph distance algorithm

Eades's original spring model only contemplates attractive forces between direct neighbors and repulsive forces between any other pair, which does not discriminate between non-adjacent nodes that are closer or farther apart. To expand on the spring model, Kamada and Kawai include the concept of an ideal distance between non-adjacent nodes based on their shortest-path distance (FRUCHTERMAN; REINGOLD, 1991).

Kamada and Kawai's algorithm models the layout generation as a minimization problem (KAMADA; KAWAI, 1989), formulating the total energy $E(L)$ of a given layout L as the sum of the energy of all of its springs (edges):

$$E(L) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} (|p_i - p_j| - l_{ij})^2 \tag{2.2}$$

In the formula above, k_{ij} is the strength of the spring modeled by the edge p_{ij}, defined as $k_{ij} = K/d_{ij}^2$, where K is a constant and d_{ij} is the length of the shortest path between i and j. l_{ij} is the ideal length of the edge p_{ij}, given as a function of the graph's diameter (shortest-path distance between the two farthest apart nodes) and the maximum desired size for the full layout (KAMADA; KAWAI, 1989).

Local minima of E can then be computed by freezing every node except for a chosen node p_m, allowing E to be written as a function of only (x_m, y_m), which can then be partially derived. The node p_m is chosen iteratively as the node with the largest Δ_m given by:

$$\Delta_m = \sqrt{(\frac{\partial E}{\partial x_m})^2 + (\frac{\partial E}{\partial y_m})^2} \tag{2.3}$$

Then, p_m is moved iteratively by a $(\delta x, \delta y)$ factor until Δ_m becomes small enough, with δx and δy given by the following linear system (HAREL; KOREN, 2000), usually solved with the Newton-Raphson method:

$$\frac{\partial^2 E}{\partial x_m^2}\delta x_m + \frac{\partial^2 E}{\partial x_m \partial y_m}\delta y_m = -\frac{\partial E}{\partial x_m} \tag{2.4}$$

$$\frac{\partial^2 E}{\partial y_m \partial x_m}\delta x_m + \frac{\partial^2 E}{\partial y_m^2}\delta y_m = -\frac{\partial E}{\partial y_m} \tag{2.5}$$

Among the common aesthetic criteria for generated layouts, this algorithm prioritizes symmetry over minimization of edge crossings, arguing that the former is as important as or more important than the latter for human readability (KAMADA; KAWAI, 1989), as shown by the following example of two representations of the same graph, one symmetrical and one planar:

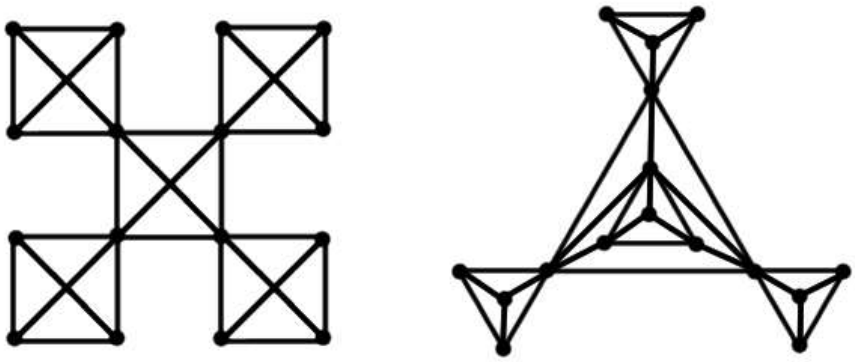

Figure 2.2 Symmetrical and planar representations of a graph (KAMADA; KAWAI, 1989).

The algorithm is also inherently suitable to represent weighted graphs due to its usage of graph distances. It was originally developed for two dimensions, but can be adapted to an n-dimensional space (HASAL; NOWAKOVA; PLATOS, 2017) with some adjustments to the formulas. Its main drawbacks are the preprocessing time and memory space required to calculate and keep the shortest-path distance between all pairs of nodes in the graph. The all-pairs shortest paths problem can be solved in sub-quadratic time using binary or Fibonacci heaps (GAJER; GOODRICH; KOBOUROV, 2000), but quadratic space is still required to store the matrix.

2.3.2 Fruchterman and Reingold's refined spring model

Fruchterman and Reingold further developed Eades's spring model by simulating concepts of particle physics, including some of the ideas by Kamada and Kawai. This method is inspired by the strong nuclear force, which is inversely proportional to the distance between the nuclei, attractive when farther apart and repulsive when closer together (FRUCHTERMAN; REINGOLD, 1991). Applying this concept to the layout generation helps prevent nodes from being drawn too close, which would negatively impact readability.

In implementation, the algorithm keeps Eades's premise of only allowing adjacent pairs to attract each other, but considers that all pairs repel each other (even if they're adjacent). It also introduces the concept of *temperature*, a dynamic parameter that limits a node's displacement on a given iteration so as the algorithm progresses and a local minimum is approached, adjustments become increasingly finer.

Algorithm 3: Fruchterman-Reingold's refined spring algorithm

Input: Graph G=(V,E), desired maximum size S of layout, K iterations

k = $\Theta(\sqrt{S/|V|})$;
for $i = 1$ **to** K **do**
  foreach v *in* V **do**
    $\overrightarrow{F_v} = 0$;
    foreach $u \neq v$ *in* V **do**

$$\Delta_{uv} = pos_v - pos_u$$
$$\overrightarrow{FRep_{uv}} = normalized(\Delta_{uv}) * \frac{k^2}{|\Delta_{uv}|}$$
$$\overrightarrow{F_v} = \overrightarrow{F_v} + \overrightarrow{FRep_{uv}}$$

    end
  end
  foreach e *in* E **do**

$$(v, u) = endpoints(e)$$
$$\Delta_{vu} = pos_v - pos_u$$
$$\overrightarrow{FAttr_{vu}} = normalized(\Delta_{vu}) * \frac{|\Delta_{uv}|^2}{k}$$
$$\overrightarrow{F_v} = \overrightarrow{F_v} - \overrightarrow{FAttr_{vu}}$$
$$\overrightarrow{F_u} = \overrightarrow{F_u} + \overrightarrow{FAttr_{vu}}$$

  end
  foreach v *in* V **do**

$$pos_v = pos_v + normalized(\overrightarrow{F_v}) * min(|\overrightarrow{F_v}|, temperature)$$

  end
  temperature = cool(temperature);
end

The dynamic parameter $k = \Theta(\sqrt{S/|V|})$ represents the optimal distance between a pair of nodes, if all nodes in V are uniformly placed in the desired space S. Further constraints can be added to the last loop if rigid adherence to S is important. The formulas of $FAttr_{uv} = |\Delta_{uv}|^2/k$ and $FRep_{uv} = k^2/|\Delta_{uv}|$ were chosen for computing efficiency

(compared to Eades's logarithmic springs) and decent avoidance of local minima, while also resembling Hooke's law (FRUCHTERMAN; REINGOLD, 1991). They also make it so the attractive and repulsive forces between the same pair of nodes cancel each other out when $|\Delta| = k$.

2.4 LARGE GRAPHS

In the scope of force-directed algorithms, there are two main problems that constrain their efficacy and efficiency as the size of the target graph grows. One of them is their time complexity: for example, each iteration of Eades's algorithm above computes $O(|V|^2)$ forces (EADES, 1984), due to taking into consideration the influence of all nodes every time. A common heuristic in n-body simulation problems, which graph drawing can be considered a subset of, is to approximate the effect of distant bodies as a single pole, decreasing the complexity of calculating interactions from $\Theta(n^2)$ to about $\Theta(n * log(n))$, depending on the approximation used (FRUCHTERMAN; REINGOLD, 1991). The idea behind approximating is that many non-connected node pairs are simply far enough apart that the force between them, for a single given pair, is insignificant. However, if the graph exhibits a structure where a cluster of nodes are comparatively close together, the force they collectively exert on a further node becomes again significant, and can be reasonably approximated for the cluster as a whole instead of calculated node by node.

The other major problem is that the function that represents the total energy (or force) of a given configuration has many local minima, an issue that grows along with the size of the graph. Attempts to minimize the issue through heuristics like simulated annealing improved the quality of embeddings in smaller graphs (though larger than the graphs initially targeted by the original force-directed algorithms), but did not achieve good results in large graphs (HADANY; HAREL, 2001).

The first force-directed algorithms to specifically tackle large graphs (>1000 nodes) pioneered the now-widespread approach of drawing the graph in stages (HADANY; HAREL, 2001), producing progressively coarser representations, or *scales*, of the full graph that can then be refined in the same manner as for a small graph. As the graph approaches its full size, the neighborhood considered for force calculations is restricted, therefore generating "locally nice" layouts. It then follows that merging "locally nice" layouts with larger "nice" layouts for each scale (coarsened representation) of the graph produces a "globally nice" layout (HAREL; KOREN, 2000). "Niceness", in this context, is a metric analogous to "readability" as described earlier; that is, fulfilling certain aesthetic criteria such as display of similarities, low clutter and minimized edge crossings, though the specific criteria prioritized depend on the implementation. This approach works well to mitigate the issues outlined further above, as it both drastically reduces the amount of node pairs to calculate forces between, and reduces the issues caused by local minima by effectively reducing the size of the graph that is relevant to a given iteration (HADANY; HAREL, 2001).

One downside of the multi-scale approach is that the quality of the final drawing is highly dependent on the quality of the embeddings generated for each scale (GAJER; GOODRICH; KOBOUROV, 2000), the computation of which is often expensive. The

majority of graph-drawing algorithms for large graphs utilize the multi-scale approach, mostly differing in the scale generation process and the force-directed algorithm chosen for local refinement.

The readability and aesthetic of the produced embedding often grows as the number of coarsening steps (scales or levels) increases (HADANY; HAREL, 2001), but the target graph itself must be considered when determining finer points of the implementation. For example, certain graphs such as sparse grids benefit from considering larger neighborhoods, but other graphs may end up losing readability of local structures in doing so (HAREL; KOREN, 2000). Different force-directed algorithms may also be employed for differently structured graphs, for the same reasons as in smaller graphs.

2.4.1 Hadany and Harel's multi-scale algorithm

Hadany and Harel define the scales of a graph $G = (V, E)$ as a sequence of k graphs $G = G_0 = (V_0, E_0), ..., G_k = (V_k, E_k)$ where $|V_{i+1}| < |V_i|$. In this algorithm, each G_{i+1} is derived from Gi by contracting edges in such a way that clusters together the pair of nodes (u, v) that delimit the contracted edge (HADANY; HAREL, 2001).

The edge (u, v) to be contracted in a given iteration j (transforming a given scale G_j into a coarser scale G_{j+1}) is chosen as to minimize the following three properties as much as possible:

- Cluster number $\rightarrow Cl(u, v) = |C_{G_j}(u)| + |C_{G_j}(v)|$ (with $C_{G_j}(u)$ representing the nodes in G_j already in the node cluster $u \in G_j$)
- Degree number $\rightarrow Deg(u, v) = max(deg_{G_j}(u), deg_{G_j}(v))$
- Homotopic number $\rightarrow Ho(u, v) = |\{w \in G_j || (w, u), (w, v) \in E_j\}|$

The cost μ associated with the edge (u, v) is calculated in relation to the three properties above so G_{j+1} captures some topological properties of G_j, despite containing significantly less nodes (HADANY; HAREL, 2001). Once a scale is generated, its local refinement (or *relaxation*, as named in the original text) works similarly to Kamada and Kawai's graph distance approach (KAMADA; KAWAI, 1989), with the energy H_i of a scale G_i formulated as:

$$H_i = \sum_{u,v \in V_i} w_{uv}(||X_i(u) - X_i(v)|| - h * l_{uv})^2 \tag{2.6}$$

In which $X(v)$, from $X : V \rightarrow R^2$, is the position of v in a configuration X that preserves and translates graph distances between pairs into their Euclidean distances, h is a constant scaling factor, l_{uv} is the graph distance between u and v and $w_{uv} = 1/d_{G_0}(u, v), u, v \in V_0$ (with $d_{G_0}(u, v)$ being the graph distance between u and v in G_0) and then recursively calculated for each scale.

Then, each scale i is refined by minimizing H_i^j (the family of energy functionals derived from H_i (HADANY; HAREL, 2001), where j denotes the size of the neighborhood

considered) for each vertex $u \in V_i$, partially deriving H_i^j with respect to $X_i(u)$ where X_i is the current configuration:

$$\frac{\partial H_i^j}{\partial X_i(u)} = \sum_{d_{G_i}(u,v) \leq j} \frac{||X_i(u) - X_i(v)||/h - l_{uv}}{||X_i(u) - X_i(v)||/h}(X_i(u) - X_i(v)) \tag{2.7}$$

Normalizing the vector given by the formula above yields a unit vector dir_u, in the direction of which $X_i(u)$ is moved until H_i^j is no longer reduced.

This algorithm is friendly to weighted graphs for the same reason as Kamada and Kawai's (KAMADA; KAWAI, 1989), that is, the usage of graph distances or shortest-path distances as a factor in the determination of the ideal Euclidean distance between a pair of nodes. Its efficacy has been shown to increase along with k (that is, the number of scales computed), in which the global structure of the graph becomes clearer as k grows (HADANY; HAREL, 2001). The original work demonstrates good results with 3 to 6 scales, depending on the graph drawn.

2.4.2 Harel and Koren's multi-scale algorithm

Harel and Koren formalize the idea behind multi-scale drawing algorithms, originally from Hadany and Harel's work (HADANY; HAREL, 2001), observing that the structural characteristics that lead to a good layout in a macro scale are characteristics of the graph as a whole with little influence from microstructures, and the inverse also holds true in which the global structure of a graph has little bearing on what constitutes a good layout at a micro scale (HAREL; KOREN, 2000). They define the notion of a "nice" layout $L*(G)$ of a graph $G = (V, E)$ as the assumed single optimal layout with respect to the aesthetic criteria desired, and demonstrate that nice layouts of a multi-scale representation $G_0, G_1, ..., G_m$ of G, with each G_i generated by locality-preserving k-clustering (k-lpc for short), induce a final layout $L*(G)$ that is both globally nice and locally nice (HAREL; KOREN, 2000).

The generation of a k-lpc can be approximated as a k-clustering/k-centers problem, which are known to be NP-hard. The method employs a 2-approximation algorithm to the k-centers problem (COFFMAN; GAREY; JOHNSON, 1996) that iteratively finds the farthest node from all the already chosen centers, choose it as a new center, then repeat until k centers are chosen. k is generated exponentially from static user-determined parameters, $|V|$, and the desired number of scales.

Once a scale G_i is generated, it can be locally refined with respect to a k-neighborhood, defined as $N_i^k(v) = \{u \in V_i | 0 \leq d_{uv} < k\}$ with d_{uv} representing the graph distance between u and v in G_i. The local refinement method chosen is based on graph distances (KAMADA; KAWAI, 1989), seeking to minimize an energy functional that relates the graph distance between the nodes to the Euclidean distance between them in the drawing (HAREL; KOREN, 2000).

This algorithm carries similar drawbacks as Kamada and Kawai's graph distance algorithm (KAMADA; KAWAI, 1989), namely the need to compute and store the all-pairs shortest distances for the whole graph. Still, in the original implementation, the majority

of running time is consumed by the local refinement stage, though the preprocessing stage becomes more significant as the graph grows. Execution complexity is described as $\Theta(|V||E|)$ with the all-pairs shortest distances being computed through breadth-first search, while space complexity is described as $\Theta(|V|^2)$ (HAREL; KOREN, 2000).

Algorithm 4: Harel and Koren's multi-scale algorithm

Input: Graph G=(V,E), parameters *threshold*, *ratio* and *neighborhoodSize*

compute all-pairs shortest distances ($d_{V \times V}$);
initialize nodes randomly;
def *k-centers*($G_i(V_i, E_i), k$)**:**
    $S = \{v\}$, for some $v \in V_i$;
    for $i = 2$ **to** k **do**
        $S = S \cup \{u\}$, for the farthest node $u \in V_i$ from S;
    end
    return S;
def *localRefinement*($G_i(V_i, E_i), d_{V_i \times V_i}, neighborhoodSize$)**:**
    foreach $v \in V_i$ **do**
        find neighborhood N_v^i of v of size *neighborhoodSize*;
        compute $pos(v)$ according to N_v^i utilizing Kamada and Kawai's algorithm;
    end
def *center*(v, i)**:**
    return center $c \in G_i(V_i, E_i)$ closest to v;
$k = threshold$;
while $k \leq |V|$ **do**
    G_k = **k-centers**($G(V, E)$, k);
    localRefinement(G_k, $d_{V_k \times V_k}$, *neighborhoodSize*);
    foreach $v \in V$ **do**
        $pos(v) = pos(center(v, k))$ + noise;
    end
    $k = k$ * *ratio*;
end

2.4.3 Gajer, Goodrich and Kobourov's multi-dimensional algorithm

As the effectiveness of multi-scale large graph drawing algorithms is largely dependent on the quality of the scales produced, one of the main problems that need to be addressed is the cost of generating said scales, which is often prohibitively expensive for most applications (GAJER; GOODRICH; KOBOUROV, 2000). Good scaled embeddings must preserve as much topological information from the original graph as possible, and many proposed algorithms are effective but not efficient. The high space requirement of earlier algorithms (HAREL; KOREN, 2000) is also undesirable.

To mitigate those issues, Gajer *et al* propose a new method of producing coarse embeddings, utilizing MIS (*maximal independent set*) filtrations (GAJER; GOODRICH; KOBOUROV, 2000). The method is based on the concept of *independent sets*, in which a

set $S \subset V$ of a graph $G = (V, E)$ is independent if none of the elements of S are adjacent in V. Therefore, a maximal independent set filtration of $G = (V, E)$ is a sequence of sets $V = V_0 \supset V_1 \supset ... \supset V_k \supset \emptyset$ such that each V_i is a maximal independent set of V_{i-1}, that is, a maximal subset of V_{i-1} for which the shortest-path distance between any two of its elements is at least (for unweighted graphs) $2^{i-1} + 1$. MIS filtrations have some advantages over other methods such as k-clustering (HAREL; KOREN, 2000), as it does not require prior computing of the all-pairs shortest distances and is naturally based on the graph's inner geometry rather than an user-defined parameter (GAJER; GOODRICH; KOBOUROV, 2000).

Algorithm 5: Gajer, Goodrich and Kobourov's maximal independent set (MIS) filtrations generation algorithm

```
Input: Graph G = (V, E)
V_0 = V;
i = 1;
while 2^i <= diameter(G) do
    V_i = v, for a random v ∈ V_{i-1};
    V_aux = V_{i-1} − {v};
    while V_aux ≠ ∅ do
        foreach u ∈ V_aux do
            if d_G(v, u) ≤ 2^i then
                V_aux = V_aux − {u};
            end
        end
        V_{i-1} = V_{i-1} ∪ {w}, for a random w ∈ V_aux;
        V_aux = V_aux − {w}
    end
    i = i + 1;
end
return sets V_0, V_1, ..., V_k;
```

Another issue raised by Gajer *et al* is that the random initialization performed by most force-directed algorithms might initially place nodes very far from their final position and therefore require extra iterations to close this distance, which can be avoided by better choosing the nodes' initial position (GAJER; KOBOUROV, 2001). Each node is only placed when its depth (the deepest filtration said node is an element of) is reached, with its initial placement computed as a function of the position of its three (assuming two-dimensional drawing) closest nodes that have already been placed. The goal is to find a point to initialize a node $t \in V_i$ such that its Euclidean distance to the three nodes $u, v, w \in V_{j<i}$ closest (in graph distance) to t is as close as possible to the graph distance between them (GAJER; GOODRICH; KOBOUROV, 2000). That is:

$$\begin{cases} (x - x_u)^2 + (y - y_u)^2 = (d_G(u, t) * k)^2 \\ (x - x_v)^2 + (y - y_v)^2 = (d_G(v, t) * k)^2 \\ (x - x_w)^2 + (y - y_w)^2 = (d_G(w, t) * k)^2 \end{cases} \tag{2.8}$$

In which k is a scaling constant. The system above may not have a solution at all, so a heuristic is often necessary. The equations can be split into three systems of two equations to yield multiple possible solutions (GAJER; GOODRICH; KOBOUROV, 2000), or t can simply be placed in the barycenter of u, v and w (GAJER; KOBOUROV, 2001); the quality of the heuristic chosen will influence the amount of refinement rounds necessary later. To start placing the nodes, the MIS filtrations can be tweaked so the smallest one has exactly three nodes (assuming two-dimensional drawing), which can then be placed following the same criterion above (GAJER; GOODRICH; KOBOUROV, 2000).

For refinement, the size of the neighborhood considered is determined dynamically as $|N_v^i| = \Theta(avgDeg(G) * |V|/|V_i|)$, to provide an upper bound of complexity. A neighborhood N_v^i can then be computed, for example, by performing a BFS in G_i starting from v and adding each node visited until N_v^i reaches the desired size. The algorithm also reintroduces the concept of *temperature* from earlier methods (FRUCHTERMAN; REINGOLD, 1991) as a scaling factor for the displacement of a node that makes adjustments progressively finer as iterations advance. A problem with the concept is that randomized initial placement makes it complicated to determine a good rate of descent for adjustments since a node might start arbitrarily close to or far from its desired position, but the heuristic of intelligent initial placement minimizes the issue (GAJER; GOODRICH; KOBOUROV, 2000).

Algorithm 6: Gajer, Goodrich and Kobourov's multi-dimensional algorithm

Input: Graph G=(V,E), parameter r

generate a filtration sequence $F : V_0 \supset V_1 \supset ... \supset V_k \supset \emptyset$;
for $i = k$ **to** 0 **do**
  compute neighborhoods $N_i(v)$, ..., $N_0(v)$;
  find initial $pos(v)$ based on the nodes already placed;
  for $j = 0$ **to** r **do**
    compute $\overrightarrow{F_i(v)}$ with respect to $N_i(v)$;
    update *temperature*;
    Δ_v = (temperature * $\overrightarrow{F_i(v)}$)
  end
  foreach $v \in V_i$ **do**
    $pos(v) = pos(v) + \Delta_v$;
  end
end

The original algorithm describes application in two-dimensional space, but it is one of the simplest algorithms to adapt for further dimensions, simply making $pos(v)$ an n-dimensional vector for any $\mathbb{R}^n$ (GAJER; GOODRICH; KOBOUROV, 2000) and considering that each node will then need $n + 1$ previously placed nodes as anchors for its initial placement (and therefore the smallest filtration must have $n + 1$ nodes). In fact, the original work demonstrates that drawing in a higher dimension and then projecting

the resulting embedding onto $\mathbb{R}^3$ or $\mathbb{R}^2$ leads to smoother drawings that also tend to better represent macro geometries (GAJER; GOODRICH; KOBOUROV, 2000).

The parameter r is a small number that can be user-defined or dynamically calculated. Typically, $5 \leq r \leq 30$ (GAJER; KOBOUROV, 2001). The calculation of $\overrightarrow{F_i(v)}$ is dependent on implementation; in the original work, it is set to a local Kamada-Kawai force vector based on graph distance (KAMADA; KAWAI, 1989) for all filtrations except the largest one ($V_0 = V$), in which it is set to a local Fruchterman-Reingold force vector (FRUCHTERMAN; REINGOLD, 1991).

Chapter 3

IMPLEMENTATION

3.1 OVERVIEW

The main requirements for a general-purpose graph visualization tool are that it offers as unconstrained a view as possible and is flexible enough to accommodate a broad range of graph types while seeking to minimize the naturally high computational cost of automated graph drawing (HAREL; KOREN, 2000).

Extant available tools are largely constrained to two-dimensional space, as exemplified by most of the graph visualization tools supported by Neo4j (JONG, 2021), currently one of the most popular database frameworks used with graph data. Two-dimensional representations are suitable enough for sparser graphs, such as trees. However, they can only produce straight-line, crossing-free embeddings for planar graphs, which was deemed too restrictive a constraint. In three-dimensional space, it is possible to generate a straight-line, crossing-free representation for any graph (DUJMOVIC; WHITESIDES, 2013).

One example of an extant 3D graph drawing tool is the GraphXR (GREEN, 2019), which is a proprietary tool for Neo4j databases, and is therefore constrained to the framework. Another example is the ForceGraph3D tool (ASTURIANO, 2022), a web component for graph visualization written in JavaScript (specifically, ThreeJS and, therefore WebGL). In targeting graph-related machine learning academic research beyond the requirements outlined above, the aim is to develop an open-source tool that is also convenient to integrate with the other tools already in use.

3.2 ENGINE

Recent advances in machine learning with graph-modeled data have mainly been in the realm of deep learning, as exemplified by the research around graph neural networks (also called graph convolutional networks) (ZHANG et al., 2019) and representation learning (HAMILTON; YING; LESKOVEC, 2018), areas in which Python reigns supreme (MOONEY, 2022). This is due to both ease of development and performance (thanks to underlying C and C++) but mainly due to the wide array of powerful ML libraries

and frameworks available, such as TensorFlow and Keras (ABADI et al., 2016). Therefore, Python 3 was the language chosen for implementation, aiming for maximum ease of integration. The graph interfacing is done with the package Networkx (HAGBERG; SCHULT; SWART, 2022), for its robust variety of available tools, and also ease of conversion to and from other common Python frameworks such as Pandas, SciPy, and NumPy.

The handling of graphics is done in OpenGL, the most widely used graphics standard in the industry, as it is flexible and offers a great degree of control to the developer. For further work beyond the proof of concept, modern OpenGL also exhibits better performance than its main competitor DirectX (EVERITT et al., 2014) and therefore offers more potential for optimization, which is desired to offset the computational cost of the drawing process itself. The better-known Python binding for OpenGL is the PyOpenGL package: it keeps almost the same syntax and commands of traditional C libraries for OpenGL (KHRONOS GROUP, 2021), which is convenient, with the downside of employing legacy OpenGL instead of modern. Nevertheless, it is sufficient for the proof of concept. The GUI shell is provided through the PyGame library as an easy plug-and-play GUI framework with a simple input capture interface that is also natively compatible with OpenGL.

For visualization proper, the camera and controls were inspired by Blender, a free open-source 3D modeling software (BLENDER FOUNDATION, 2022). OpenGL in and of itself does not include the concept of a "camera", so the term here is more of a shorthand to abstract the operations performed on the MVP (model-view-projection) matrix that actually controls the world space to screen space transformation in OpenGL (KHRONOS GROUP, 2021). To maximize flexibility in visualization, the camera is a fusion of a first-person free-flying camera and an orbital camera with a dynamic origin, allowing the user to navigate the generated graph drawing at will and also inspect any part of it freely. Full freedom of movement, depending on how it's implemented, often incurs issues such as gimbal lock, a phenomenon in which two or more of the rotation axes align with each other and cause the loss of a whole dimension of movement (PERUMAL, 2011). To avoid this problem, the camera movement is mostly implemented with quaternions. Quaternions are a type of four-dimensional vectorial object that can represent and apply three-dimensional rotations to vectors, require less space than rotation matrices, and are less subject to issues normally associated with Euler angles, like the aforementioned gimbal lock (GOLDMAN, 2011).

The spatial calculations are done almost entirely with vectors. For that, many OpenGL applications utilize the OpenGL Mathematics package, or glm (RICCIO, 2017), which provides a robust mathematical library that is based on the GLSL (OpenGL Shading Language) specification and is suitable for work involving vectors, matrices, and similar structures. In Python, PyGLM (Zuzu-Typ, 2022) offers a fast, C++-built binding for the glm library that is able to efficiently process not only the vectors and matrices naturally involved in three-dimensional digital drawing but also easily compute quaternion operations required by the camera.

3.2.1 Program structure

The program takes as input a JSON file generated by Networkx's `json_graph.node_link_data()` method (HAGBERG; SCHULT; SWART, 2022) from a Networkx `Graph` or `DiGraph` object, along with parameters to choose a drawing method and, if applicable, the number of iterations. The command-line execution call is as follows:

```
python3 main.py -f <filepath> [-i <iterations>][-m <model>][-t <target>]
```

In which:

- `filepath`: relative path to the JSON file containing the graph *(required)*
- `iterations`: number of iterations the drawing algorithm will perform, for spring and barycentric drawers *(default 100)*
- `model`: drawing algorithm to use (currently: random, barycentric, spring, multiscale) *(default spring)*
- `target`: whether to render an extra point in the screen showing the specific coordinate the camera is looking at, useful to guide orbital panning (Y/N) *(default N)*

The program then initializes a `GraphDrawer` object according to the model chosen and a `Camera` object to manage camera movement. The PyGame window is initialized and receives inputs from the mouse while it's in focus. Right-clicking and dragging moves the point of view along the plane represented by the window itself, rolling the mouse wheel moves the point of view along the view vector (that is, forward and backward), and clicking and dragging the mouse wheel moves the point of view orbitally around the camera's current target. This is to allow close inspection of microstructures of larger graphs as well as macrostructures, which is often lacking in most static representations without a dynamic point of view (GAJER; GOODRICH; KOBOUROV, 2000).

The drawer classes add some extra attributes to the nodes, namely `GV_position` and `GV_color`, which are then used by the main `GraphViewer` class in its rendering loop. `GV_position` is a 3-tuple (x, y, z) of floats indicating the Euclidean position of the node in relation to the global origin $(0.0, 0.0, 0.0)$. `GV_color` is a 3-tuple (r, g, b) of floats with $0.0 \leq r, g, b \leq 1.0$ indicating the RGB color in which OpenGL should render the node.

Since all the algorithms presented earlier are composed of preprocessing and/or initialization stages to start and then iterative refinement stages, it is possible to allow the user to watch the graph drawing process as it happens by tying each iteration of the refinement to a frame, separating the iterative section of the algorithm into a method that can be continuously called by the main rendering loop. To achieve this, the abstract class `DrawerInterface`, which all the drawer classes inherit from, requires the implementation of the `initialize()` and `runLoop()` methods for its children. The `initialize()` method is run before the GUI is initialized and encapsulates any preprocessing stages required by the drawing algorithm (such as random node initialization, generation of neighborhoods, computation of the all-pairs shortest distances, etc), while `runLoop()`

contains the algorithm's drawing loop itself and is called on each frame (though the *iterations* parameter is checked to decide whether to actually execute the loop).

3.2.2 Drawing small graphs

For drawing small graphs in $\mathbb{R}^3$, the program implements Eades's spring model (EADES, 1984) as follows, mostly just adapting it to the `DrawerInterface` class's structure:

Algorithm 7: Implementation of Eades's spring algorithm

Input: Graph G=(V,E), K iterations, array of constants $c = [c_1, c_2, c_3, c_4]$

def *initialize*$(G(V, E))$**:**
  initialize all $v \in V$ randomly in $\mathbb{R}^3$;
def *runLoop*$(G(V, E))$**:**
  if $K > 0$ **then**
    foreach $v \in V$ **do**
      $\overrightarrow{F_v} = (0.0, 0.0, 0.0)$;
      foreach $u \neq v \in V$ **do**
        if $(u, v) \in E$ **then**

$$\overrightarrow{F_{uv}} = c_1 * log(\frac{||\overrightarrow{pos[u]} - \overrightarrow{pos[v]}||}{c_2}) * normalized(\overrightarrow{pos[u]} - \overrightarrow{pos[v]})$$

        end
        else

$$\overrightarrow{F_{uv}} = \frac{c_3}{||\overrightarrow{pos[v]} - \overrightarrow{pos[u]}||^2} * normalized(\overrightarrow{pos[v]} - \overrightarrow{pos[u]})$$

        end
        $\overrightarrow{F_v} = \overrightarrow{F_v} + c_4 * \overrightarrow{F_{uv}}$;
      end
      $pos[v] = pos[v] + \overrightarrow{F_v}$;
    end
    $K = K - 1$;
  end

This drawer can be chosen by passing the argument `-m spring` to the program. It utilizes the constants suggested by the original work, in which $c_1 = c_2 = c_3 = 1$ and $c_4 = 0.1$ (KOBOUROV, 2013). Like the original algorithm (EADES, 1984), it assumes the graph is unweighted, though an adaptation may be made so that the array of constants c reflects edge weights and graph distances instead of being static.

An implementation of the barycentric algorithm (TUTTE, 1963) is also included, to show that despite being geared towards $\mathbb{R}^3$, the existing setup can also produce flat, $\mathbb{R}^2$-

like drawings. However, one of the main limitations of the barycentric algorithm is the need for the user to provide a previously known face of the graph (KOBOUROV, 2013). The program checks for the existence of a `GV_BarycentricFixedVertices` attribute in the graph that should contain a Python list of node IDs and uses it as the starting polygon if it exists. If it does not exist, the drawer computes the longest (unweighted) cycle in the graph using depth-first search and uses it instead; this heuristic does not guarantee that the cycle found is a face, and therefore does not guarantee a crossing-free drawing, but was chosen to allow for enough room inside the starting polygon so the drawing can hopefully still be readable despite the crossings. Like the spring algorithm further up, the barycentric algorithm is implemented faithfully to the original, only adapted for the method structure expected by the `DrawerInterface` class. This drawer can be chosen by passing the argument `-m barycentric` to the program.

Algorithm 8: Implementation of Tutte's barycentric algorithm

Input: Graph $G = (V, E)$, K iterations, desired drawing radius R, a cycle $C \subset V$ that represents one of the drawing's faces

def *initialize*$(G(V, E))$**:**
  if C is not provided, set C to the largest unweighted cycle in G;
  `/* place nodes in` C `according to a regular polygon with` $|C|$ `sides and radius` R `*/`
  $i = 0$;
  foreach $v \in C$ **do**
    $theta = 2\pi * (i/|C|)$;
    $pos[v] = (R * cos(theta), R * sin(theta))$;
    $i = i + 1$;
  end
def *runLoop*$(G(V, E))$**:**
  if $K > 0$ **then**
    foreach $v \in V - C$ **do**
      N_v = all nodes u such that $(u, v) \in E$;
      $pos[v] = (\sum_{u \in N_v} pos[u]) * \frac{1}{|N_v|}$
    end
  end
  $K = K - 1$;

3.2.3 Drawing large graphs

One of the challenges faced by implementing a visualization tool friendly to large graphs in Python is that many of the algorithms require solving non-linear systems, such as the algorithms based on graph distance (HAREL; KOREN, 2000). In C++, the GSL (GNU Scientific Library) package offers efficient multidimensional root finding for non-linear systems (THE GSL TEAM, 2021). In Python, the SymPy package is often recommended, but its performance suffers slightly due in part to its pure Python imple-

mentation (SYMPY DEVELOPMENT TEAM, 2021), which would become an issue because of the sheer amount of systems the program would need to solve. SciPy's `scipy.optimize.fsolve` is considered for future work, as it is a Python wrapper for fast Fortran routines (THE SCIPY COMMUNITY, 2022).

For the proof of concept, the algorithm chosen for large graphs was Gajer, Goodrich, and Kobourov's multidimensional algorithm (GAJER; GOODRICH; KOBOUROV, 2000), with the intention to adapt it for weighted graphs. To do so, the preprocessing stage calculates the all-pairs shortest distances, which takes a few seconds for a ≈1500 node graph with Networkx's `all_pairs_dijkstra_path_length()`. The computation of MIS filtrations, however, still does not take into consideration the edge weights, as it is meant to be a uniformly distributed partition of the set of nodes based on whether the edges exist at all. The graph distances are used for the initial placement of the deepest filtration, as well as for calculating local forces in the refinement stages. In using this model, the program expects all edges to have a positive `weight` attribute; for unweighted graphs, the attribute can just be set to 1.

The implementation draws in $\mathbb{R}^4$ and projects the coordinates down to $\mathbb{R}^3$ (GAJER; GOODRICH; KOBOUROV, 2000). To do so, the program creates an extra node attribute `GV_position_R4` as a 4-tuple (x, y, z, w) of floats and utilizes it as a four-dimensional position vector for all its spatial calculations. The four-dimensional vector is then projected to three dimensions utilizing the Gram-Schmidt orthogonalization process and the projection function described in the original work (GAJER; GOODRICH; KOBOUROV, 2000), implemented with NumPy's `linalg.qr` function and PyGLM vector operations. The projected three-dimensional position vector is then kept in the `GV_position` node attribute for the rendering loop to access. To be able to draw in $\mathbb{R}^4$, the program also begins by placing five nodes instead of three, and considers five neighbors instead of three when computing the initial placement of a new node.

In the refinement stage, for all filtrations except the largest one (where $V_0 = V$), the displacement of a node v is set to a local Kamada-Kawai-inspired force vector, with respect to a neighborhood $N_i(v)$. For the largest filtration where $V_0 = V$, the displacement of a node v is set to a local Fruchterman-Reingold force vector instead, with respect to the neighborhood $N_0(v)$ (GAJER; KOBOUROV, 2001). This is done to take advantage of the Fruchterman-Reingold's method's modeled repulsive forces between all nodes, to enforce a minimum distance between nodes and avoid them being drawn right on top of each other (FRUCHTERMAN; REINGOLD, 1991). The forces are calculated as follows:

$$\overrightarrow{F_{KK}(v)} = \sum_{u \in N_i(v)} \left(\frac{d_{\mathbb{R}^n}(u,v)}{d_G(u,v) * L^2} - 1\right)(pos[u] - pos[v]) \tag{3.1}$$

$$\overrightarrow{F_{FR}(v)} = \sum_{u \to (u,v) \in E} \frac{d_{\mathbb{R}^n}(u,v)^2}{d_G(u,v)^2}(pos[u] - pos[v]) + \sum_{u \in N_i(v)} \frac{d_G(u,v)^2}{d_{\mathbb{R}^n}(u,v)^2}(pos[v] - pos[u]) \tag{3.2}$$

In both equations, $d_G(u,v)$ is the graph distance between u and v, while $d_{\mathbb{R}^n}(u,v)$ is the Euclidean distance between the current positions of u and v. The order of subtraction

of $pos[u]$ and $pos[v]$ is important to determine the direction of the force. The constant L is the ideal length of an edge with weight 1; for a bounded frame, it can be calculated as a function of the available space and the graph's diameter (FRUCHTERMAN; REINGOLD, 1991). Since we are not dealing with a bounded frame, L was set to a fixed constant.

In the original work, the Fruchterman-Reingold force vector utilizes the same constant L in place of the graph distance d_G (GAJER; KOBOUROV, 2001). In the Fruchterman-Reingold algorithm, which is developed for unweighted graphs, this is done so the attraction and repulsion forces between neighbors cancel each other out when the Euclidean distance between an adjacent pair is equal to this ideal edge length L (FRUCHTERMAN; REINGOLD, 1991). So, to adapt it to a weighted graph, the program experiments with utilizing the graph distance as the ideal edge length.

The neighborhood size $|N_i(v)|$ for a given filtration G_i is determined as (GAJER; GOODRICH; KOBOUROV, 2000):

$$nbrs(i) = c * (\frac{\sum_{v \in V} degree(v)}{|V_i|}) \tag{3·3}$$

In which c is a constant. Increasing the value of c slightly improves the quality of the embedding, but also increases the complexity of calculations. To populate $N_i(v)$, since the program already has the all-pairs shortest distances computed, it's trivial to collect the $nbrs(i)$ closest nodes to v in V_i instead of employing BFS like in the original work (GAJER; GOODRICH; KOBOUROV, 2000). If $nbrs(i) > |V_i|$, simply add all of V_i to $N_i(v)$, as the $nbrs(i)$ function is just meant to provide an upper bound of complexity (GAJER; GOODRICH; KOBOUROV, 2000).

Algorithm 9: Implementation of Gajer, Goodrich and Kobourov's multi-dimensional algorithm (initialization)

Input: Graph $G = (V, E)$
1 **def** *initialize(*$G(V, E)$*, parameter L for the ideal edge length with weight 1)*:
2 compute the all-pairs shortest distances;
3 generate filtrations $V = V_0 \supset ... \supset V_k \supset \emptyset$;
4 alter the smallest filtrations until $|V_k| = 5$;
5 **foreach** $v \in V$ **do**
6 **for** $i = 0$ **to** *largest i such that* $v \in V_i$ **do**
7 $N_i(v) =$ closest $min(nbrs(i), |V_{i+1}|)$ nodes to v that belong to V_{i+1};
8 **end**
9 **end**
10 build $\mathbb{R}^4$ space;
11 $r = 0$; $i = k$;

This drawer does not take the `-i <iterations>` parameter from the command line. Instead, it receives two parameters for the minimum and maximum amount of rounds, and implements a function `rounds(i)` to compute the number of rounds per filtration, so

the number of rounds in the deepest filtration is close to the minimum and the number of rounds in the largest filtration is close to the maximum, increasing logarithmically.

Every operation done with the position of a node v uses its four-dimensional position vector $pos\mathbb{R}^4[v]$, and every assignment to $pos\mathbb{R}^4[v]$ is followed by computing the projection of $pos\mathbb{R}^4[v]$ onto $\mathbb{R}^3$, with the result stored in the `GV_position` attribute for rendering.

Algorithm 10: Implementation of Gajer, Goodrich and Kobourov's multi-dimensional algorithm (drawing loop)

Input: Graph $G = (V, E)$, parameter L for the ideal edge length with weight 1

def *runLoop(*$G(V, E)$*)*:
  if $r == 0$ *and* $i \geq 0$ **then**
    runIteration($G(V, E)$, i); r = rounds(i); $i = i - 1$;
  end
  else if $r \geq 0$ **then**
    runRound($G(V, E)$, $i + 1$, r); $r = r - 1$;
  end
def *runIteration(*$G(V, E)$*,* i*)*:
  if $i == k$ **then**
    place the 5 nodes in V_k randomly;
    refine their positions utilizing the $\overrightarrow{F_{FR}(v)}$ formula in 3.2;
  end
  else
    foreach $v \in V_i - V_{i+1}$ **do**
      a, b, c, d, e = closest 5 nodes to v that belong to V_{i+1};
      $pos\mathbb{R}^4[v] = (pos\mathbb{R}^4[a] + pos\mathbb{R}^4[b] + pos\mathbb{R}^4[c] + pos\mathbb{R}^4[d] + pos\mathbb{R}^4[e])/5$;
    end
  end
def *runRound(*$G(V, E)$*,* i*,* r*)*:
  foreach $v \in V_i$ **do**
    if $i \geq 0$ **then**
      compute $\overrightarrow{F(v)} = \overrightarrow{F_{KK}(v)}$ (formula 3.1), with respect to $N_i(v)$;
    end
    else
      compute $\overrightarrow{F(v)} = \overrightarrow{F_{FR}(v)}$ (formula 3.2), with respect to $N_i(v)$;
    end
    $temperature[v] = heat(v)$;
    $pos\mathbb{R}^4[v] = pos\mathbb{R}^4[v] + (normalize(\overrightarrow{F(v)}) * temperature[v])$;
  end

The parameter $temperature[v]$ is initially set to $L/6$. After, it is updated by the function `heat(v)` with every displacement calculation, based on the previous behavior of the node (GAJER; KOBOUROV, 2001). This drawer expects every edge to have an

edge attribute `weight` and can be chosen by passing the argument `-m multi-scale` to the program.

Chapter 4

EXPERIMENTS

4.1 GENERAL SMALL GRAPHS

The spring algorithm (EADES, 1984) was able to handle graphs up to a few hundred nodes, though with increasing execution time. Some issues were noted with the model getting caught in local energy minima. Since it initializes all nodes randomly to start with, and therefore a node's initial position relative to other nodes is completely arbitrary, situations arise in which a node might need to "move past" another node to reach its best possible position, characterizing the need to temporarily accept a worse layout to be able to reach a better layout than the previous configuration (FRUCHTERMAN; REINGOLD, 1991). This behavior is one of the heuristics used in simulated annealing to avoid local minima (BERTSIMAS; TSITSIKLIS, 1993), usually called *hill climbing*. The spring model mostly behaves as a greedy algorithm and, therefore, often becomes stuck in local minima, which was most visible on cylinders and tori as twist-like distortions.

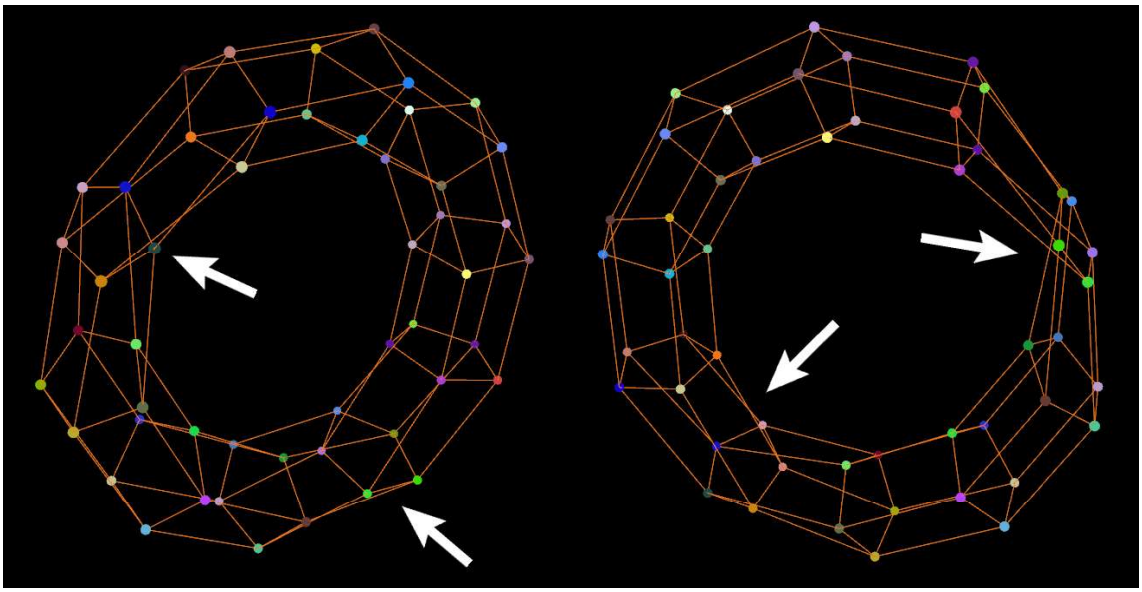

Figure 4.1 A 50-node torus, drawn by the spring algorithm, showing twisting.

In fact, the tori showed twists in all executions. The consistently even number of twists shows this was likely not caused by issues with the graph, as deliberately connecting the torus upside-down (like a mobius strip) always wielded an odd number of twists. Letting the algorithm run for an unbounded number of iterations did not remove the twists. These results lend credence to the later proposal of aiming to intelligently place nodes from the start (GAJER; GOODRICH; KOBOUROV, 2000). Two 50-node tori drawn by the spring algorithm are shown in figures 4.1 and 4.2, both having taken about 150 iterations to reach the pictured state. A 200-node torus is shown in figure 4.3 after about 400 iterations; note how the twists are more or less noticeable depending on perspective.

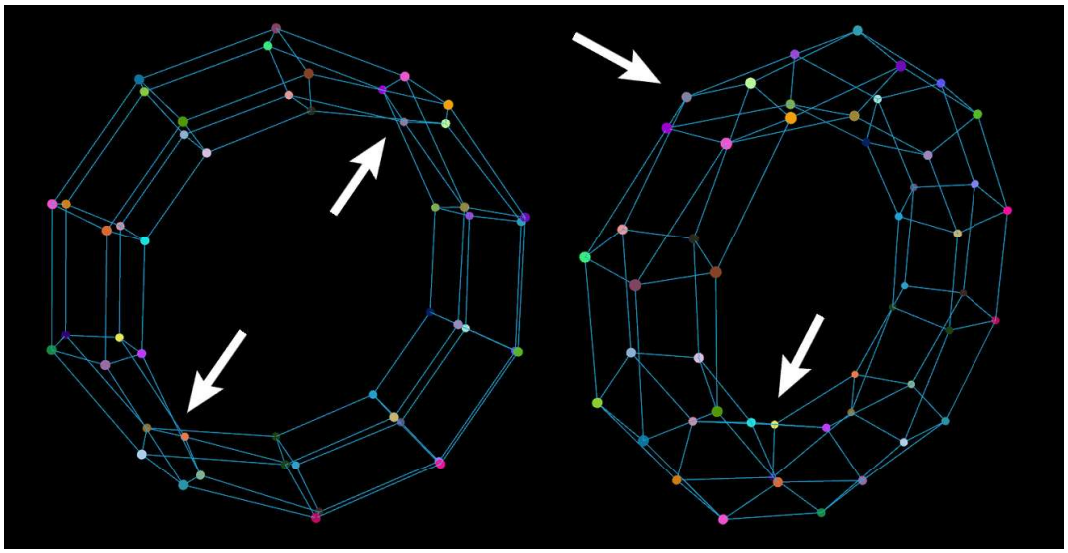

Figure 4.2 Another view of the 50-node torus from figure 4.1.

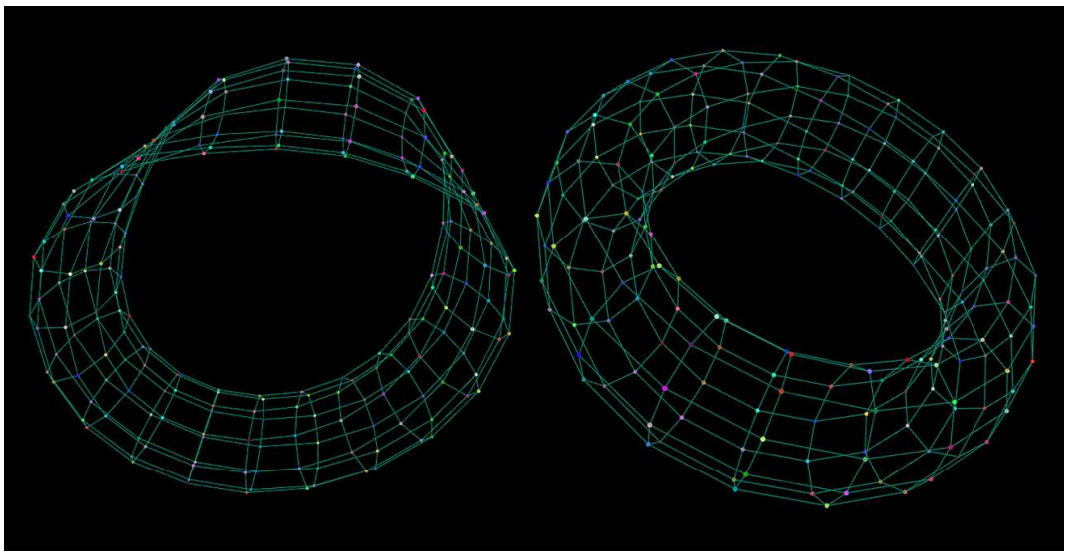

Figure 4.3 A 200-node torus drawn by the spring algorithm, showing how the twists become more or less noticeable based on perspective.

A 200-node cylinder is pictured in figure 4.4 after 300 iterations, also exhibiting a twist. The curved shape of the cylinder is a product of the algorithm's repulsion force based on the inverse square of the Euclidean distance, as it makes it difficult for a graph that is "long" in shape (such as a cylinder) to "spread out". The cylinder continued to slowly straighten when the program was allowed to run for an unbounded number of iterations, though the possibility of it eventually straightening out completely is dependent on whether its size would allow it to happen before the float result from the calculation $c/distance^2$ (see algorithms 2 and 7) becomes indistinguishable from zero in machine representation.

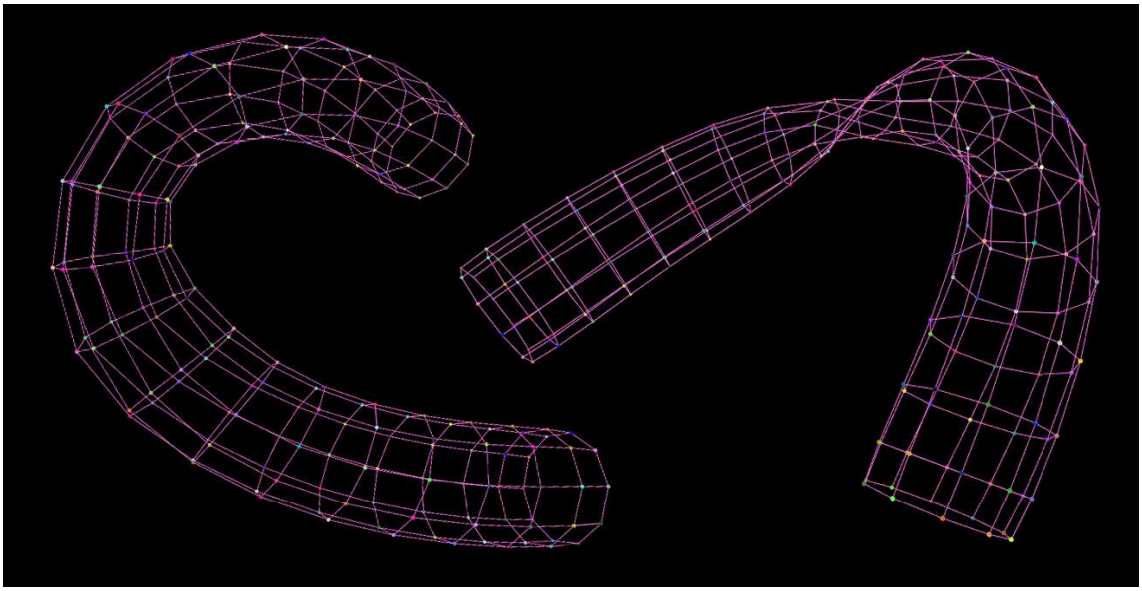

Figure 4.4 A 200-node cylinder drawn by the spring algorithm. Note the twist.

A similar behavior to the cylinder was observed on planar graphs like meshes, exemplified in figure 4.5 at ∼100 and ∼300 iterations. In cases like this, thanks to the implementation of the program tying the algorithm's drawing loop to the rendering loop as described in the previous chapter, it is possible to watch the mesh unfurl as the algorithm progresses. The 100-node, 10x10 mesh shown in figure 4.5 eventually reached uniformity in edge lengths (since the spring algorithm, as-is, is developed for unweighted graphs (EADES, 1984)); the apparent distortion in the second part of the figure is caused by perspective.

To demonstrate application of the model on a more general graph, a network representing the co-occurrence of characters in the book Les Misérables by Victor Hugo was used, with 77 nodes and 254 edges, in which nodes represent characters and edges indicate a pair of characters appearing in the same chapter (KUNEGIS, 2018). The drawing is shown in figure 4.6. It shows the algorithm's tendency to keep clusters away from each other due to the concentration of repelling forces (FRUCHTERMAN; REINGOLD, 1991). This helps avoid cluttering, but, in graphs with high degree variance, it might lead to overly long edges at the points of connection between clusters, as shown.

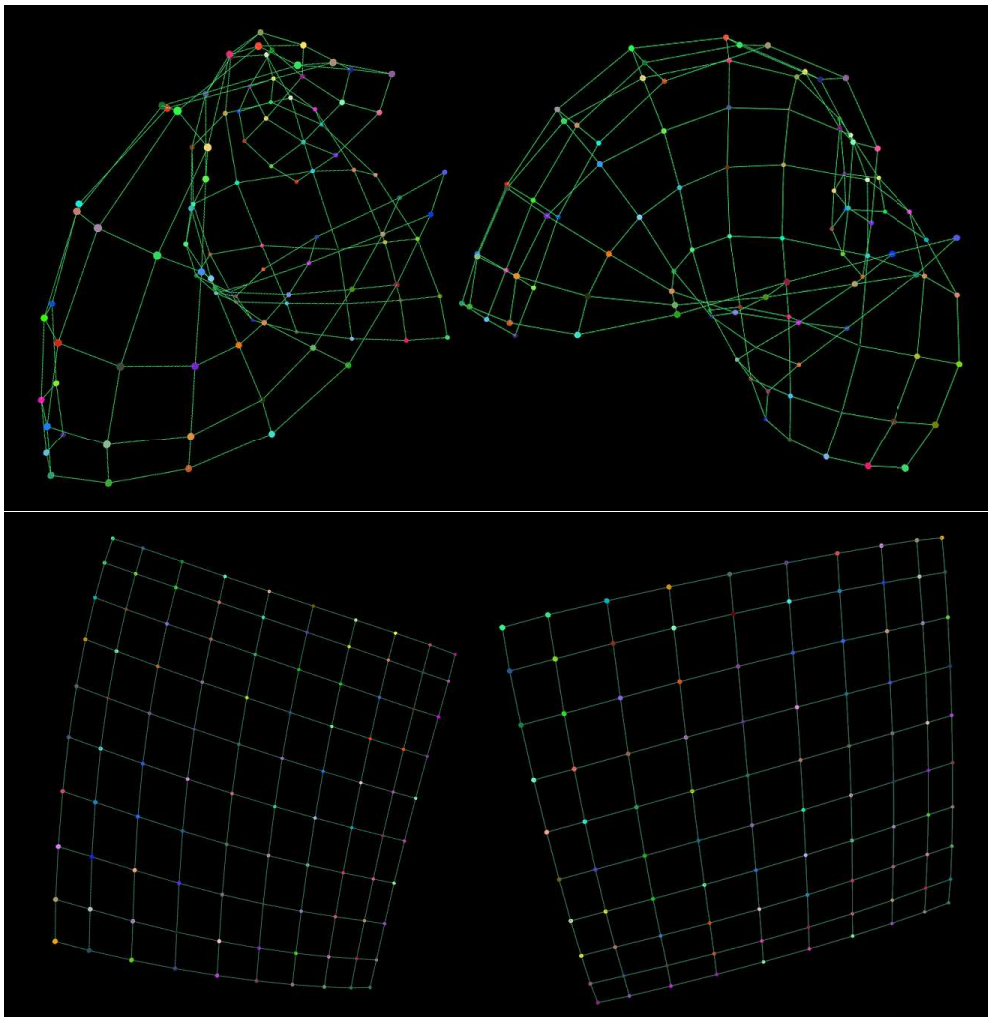

Figure 4.5 A 10x10 mesh drawn by the spring algorithm at different stages of the process.

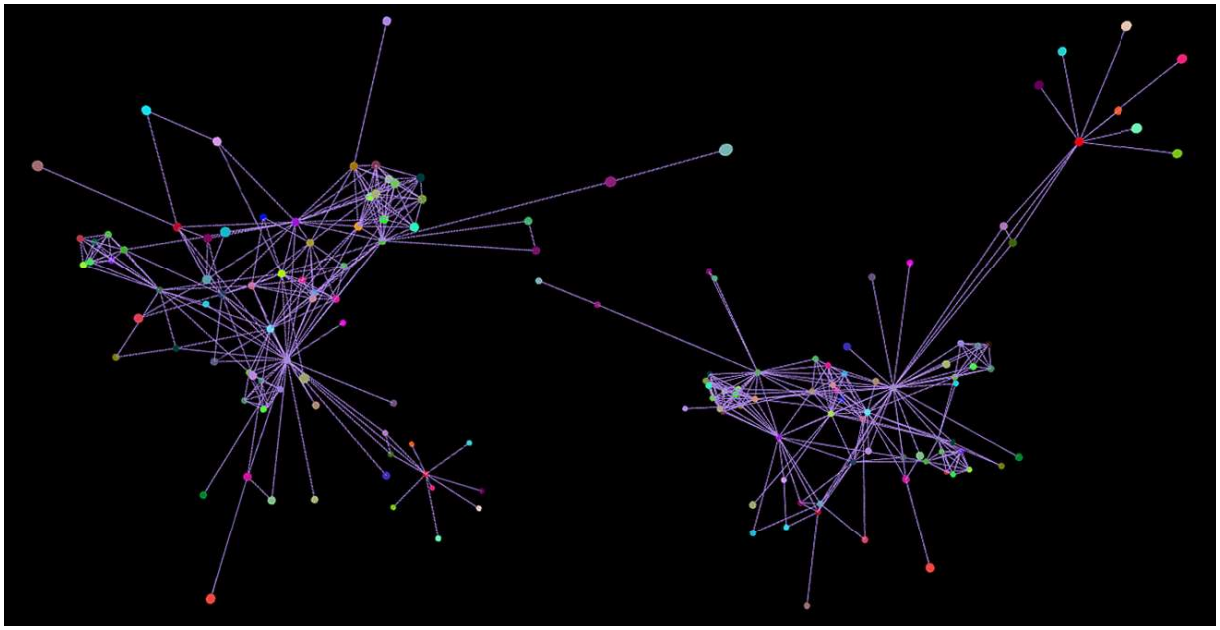

Figure 4.6 Two views of the Les Misérables co-occurence network (KUNEGIS, 2018), drawn by the spring algorithm.

4.1.1 Planar graphs

The program allows rendering two-dimensional representations in the three-dimensional space by making all the points that represent the nodes co-planar. The implementation of the barycentric algorithm (TUTTE, 1963) does this by setting the z coordinate of all points to zero (in OpenGL, the z axis is the axis that goes perpendicularly "into" the screen). This way, placing the nodes in the barycenter of their already-placed neighbors keeps their z coordinates at 0, and the drawing stays flat (see algorithms 1 and 8). The same idea can be applied to any two-dimensional drawing algorithm. For adapting iterative force-directed algorithms to 2D, the algorithm can be minorly adapted to ignore the z coordinate of the force vector and achieve the same result.

The barycentric algorithm has lower complexity than the spring algorithm (EADES, 1984) due to naturally being neighborhood-restricted (HADANY; HAREL, 2001), so it can theoretically handle graphs in the same order of magnitude, at least regarding execution time; the 101-node circular mesh shown in figure 4.7 took only about 30 iterations to reach the pictured state.

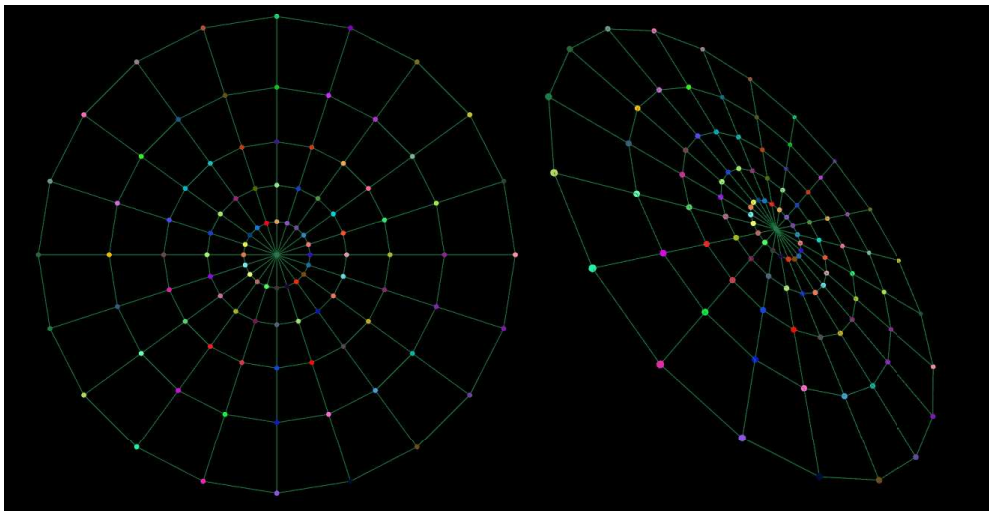

Figure 4.7 A 101-node circular mesh drawn by the barycentric algorithm.

The algorithm's main limitation in relation to graph size is actually the size of the resulting drawing, which can have up to exponential area (EADES; GARVAN, 1995). This is noticeable in figure 4.8, showing a 201-node circular mesh with the same amount of sides as the 101-node circular mesh in figure 4.7, but with twice as many rows. The drawing only took about 50 iterations to reach the pictured state, but the innermost rows are hard to make out (for reference, the nodes in figures 4.7 and 4.8 are rendered in the same size). Figure 4.8 also exemplifies the usefulness of the camera's full freedom of movement, as it allows the user to inspect the smaller center structures that are not readable in the full drawing.

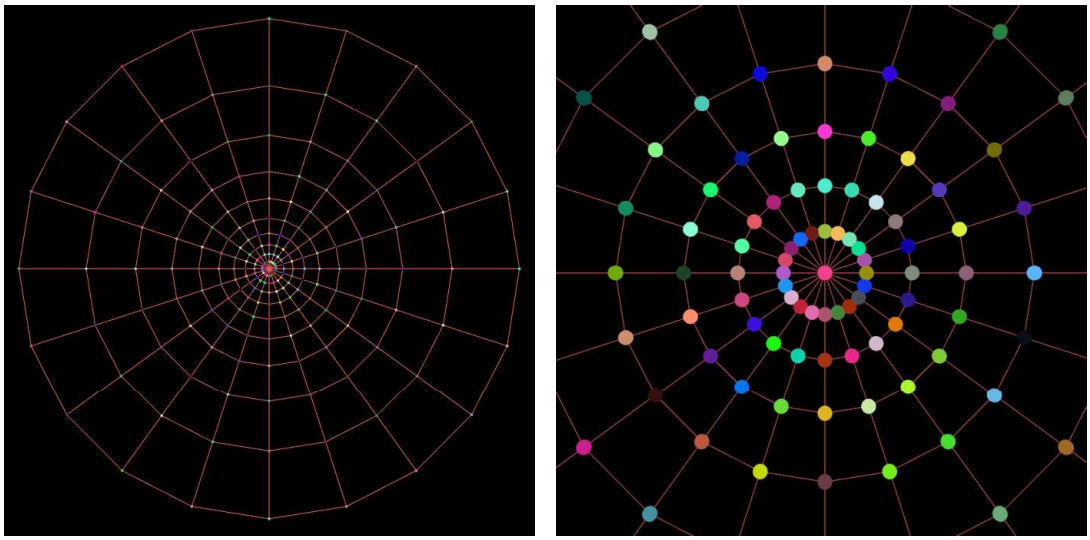

Figure 4.8 A 201-node circular mesh drawn by the barycentric algorithm, observed at different zoom distances.

4.2 GENERAL LARGE GRAPHS

For larger graphs (over a few hundred nodes), the chosen algorithm was Gajer, Goodrich and Kobourov's algorithm (GAJER; GOODRICH; KOBOUROV, 2000), from now on referred to as the multi-dimensional algorithm, which was adapted to take edge weights into consideration (see algorithms 9 and 10).

The main point noted by this experiment was the importance of the chosen heuristic to intelligently choose initial node placements, which was already mentioned in the original work (GAJER; GOODRICH; KOBOUROV, 2000), but proved itself even more impactful than expected. In earlier experiments, the drawing was still done in $\mathbb{R}^3$ and therefore the smallest filtration held only 4 nodes, which made it easier to place 3 of them in a triangle according to their graph distances, and then place the last one on the axis perpendicular to the triangle's plane. This sometimes wielded recognizable results, as shown by the 1500-node cylinders in figure 4.9. This approach was also restricted by the fact that it was often not possible to draw a triangle from the graph distances between the starting nodes, and the quality of the final drawing was very inconsistent. Intelligently placing nodes on subsequent iterations was done following the heuristic proposed by the original authors (GAJER; KOBOUROV, 2001) of placing them in the barycenter of their already placed neighbors (see sections 2.4.3 and 3.2.3); a more precise heuristic will likely yield better results.

The algorithm was then set to draw in $\mathbb{R}^4$, to analyze if it would indeed produce smoother drawings (GAJER; GOODRICH; KOBOUROV, 2000). However, increasing dimensions also increases the number of nodes in the deepest filtration (see 2.4.3), making it more challenging to compute an acceptable starting configuration. In the end, for the proof of concept, the heuristic chosen was to place those five starting nodes utilizing Fruchterman and Reingold's refined spring algorithm (FRUCHTERMAN; REINGOLD, 1991), adapted to have the ideal edge length reflect edge weights. This was fast to

compute as it only had to place a very small number of nodes, and achieved Euclidean distances very close to the ideal graph distances. Iterations after the first still initially placed new nodes in the barycenter of their neighbors (GAJER; KOBOUROV, 2001). This approach yielded recognizable but quite twisted shapes, shown by the 1500-node cylinders and torus in figure 4.10.

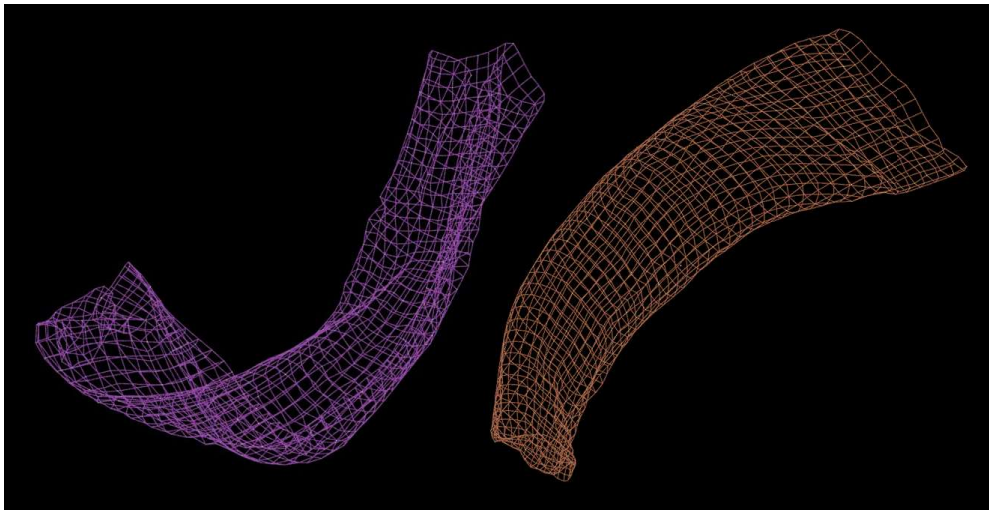

Figure 4.9 Two 1500-node cylinders drawn in $\mathbb{R}^3$ by the multi-dimensional algorithm, with the simpler heuristic for initially placing the deepest filtration.

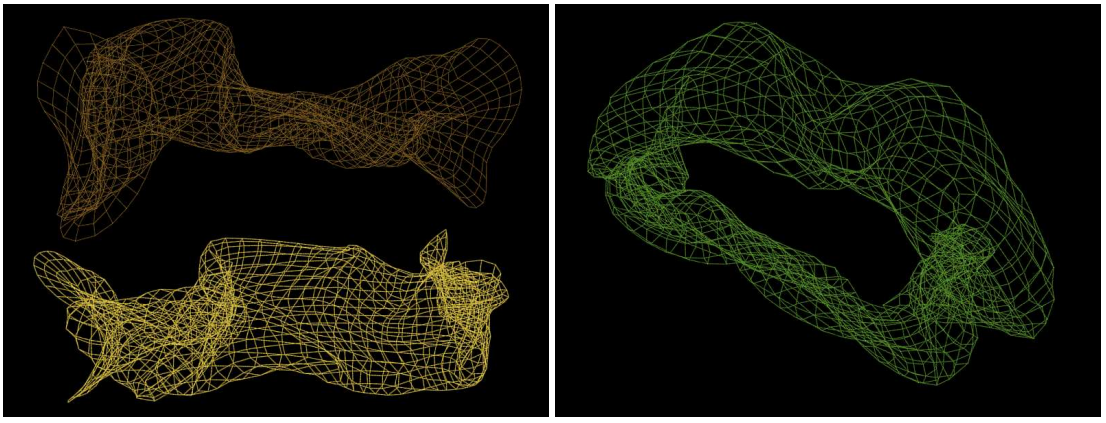

Figure 4.10 Two 1500-node cylinders and a 1500-node torus drawn in $\mathbb{R}^4$ and projected to $\mathbb{R}^3$ by the multi-dimensional algorithm, with the deepest filtration placed in a force-directed manner.

The program then attempted to combine the more effective initial placement by refined spring algorithm seen in figure 4.10, and the better overall shape from drawing in $\mathbb{R}^3$ seen in figure 4.9. At first, this produced worse drawings, until the function `nbrs(i)` (see equation 3.3 and section 3.2.3) was tweaked to consider larger neighborhoods in local force calculations. It took neighborhoods four times larger for the direct $\mathbb{R}^3$ drawing of

the 1500-node torus to be comparable to the $\mathbb{R}^4$ drawing in figure 4.10, with great cost to execution time. The torus is shown in figure 4.11. The 1500-cylinder showed very little improvement even with the larger neighborhood and therefore was not included.

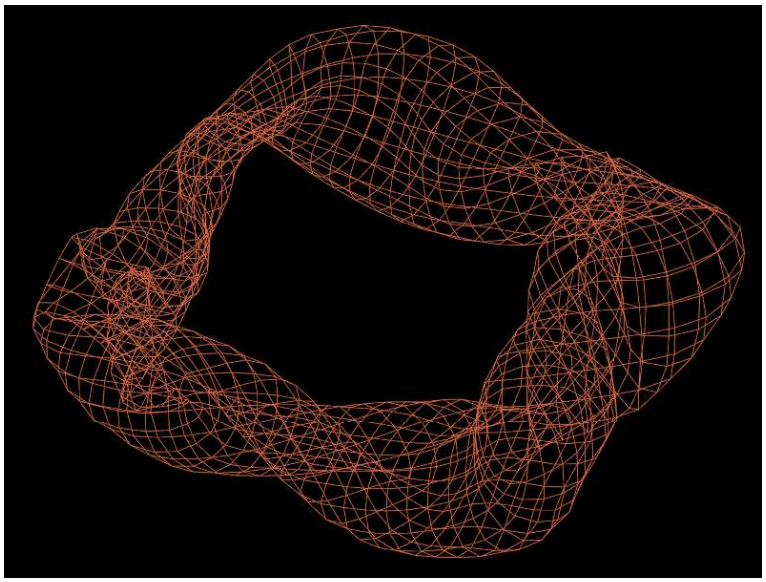

Figure 4.11 A 1500-node torus drawn directly in $\mathbb{R}^3$ by the multi-dimensional algorithm, considering neighborhoods four times larger than the ones in 4.10.

Attempts to improve the $\mathbb{R}^4$ drawings by also increasing neighborhood size did not yield visible improvement before the increased computational complexity became too costly. Increasing the number of refinement iterations helped, but had diminishing returns due to the cooling heuristic as it is currently implemented. In the end, the implementation chosen was the one that produced figure 4.10: drawing in $\mathbb{R}^4$, placing the deepest filtration in a force-directed manner, and considering smaller neighborhoods.

4.3 PROTEIN INTERACTION DATA

The main motivation behind the development of this proof of concept was to provide a convenient visualization tool for studying protein interactions and relationships, in the context of researching the severity of hemophilia in patients (LOPES et al., 2021b). In this research, proteins and amino acids are modeled as graph data, which is then supplied to a machine learning model to predict outcomes (LOPES et al., 2021a). The graphs generated include multiple attributes containing information about the proteins, but the visualization tool is more interested in the structure of the graph, other than using the labels as the criterion to color nodes.

The raw data makes up a connected, undirected, weighted graph with 1317 nodes and 4815 edges, which was first preprocessed to isolate the relevant fields (the node IDs, the configuration of the edges including their weights, and the label `severity`, which is a binary attribute that was then translated into the `GV_color` attribute used by the tool), which was done using Pandas data frames. Then, transforming the data frames

into the Networkx graph object expected by the tool was easily done with Networkx's `from_pandas_edgelist` function.

The full graph is well connected, with an average unweighted degree of ∼7.3, including 300 nodes with an unweighted degree of at least 10. On the one hand, this makes it a good candidate for force-directed drawing in general (DI BATTISTA et al., 1999), as the abundance of edges means more forces at play that make reaching static equilibrium happen faster. On the other, the edges do not follow a strict pattern, leading to a lot of unavoidable interlocking that negatively impacts readability.

It is hard to convey through two-dimensional images the three-dimensional form of an object that does not have an easily recognizable shape like a cylinder, mesh or torus, but figure 4.12 shows some views of the 3D embedding built from the full graph by the multi-dimensional algorithm (GAJER; GOODRICH; KOBOUROV, 2000), drawn in $\mathbb{R}^4$ and projected to $\mathbb{R}^3$ (see section 2.4.3 and algorithms 9 and 10). The nodes colored light blue have the `severity` label set to `low`, while the nodes colored orange have the `severity` label set to `high`. White nodes do not have a `severity` label.

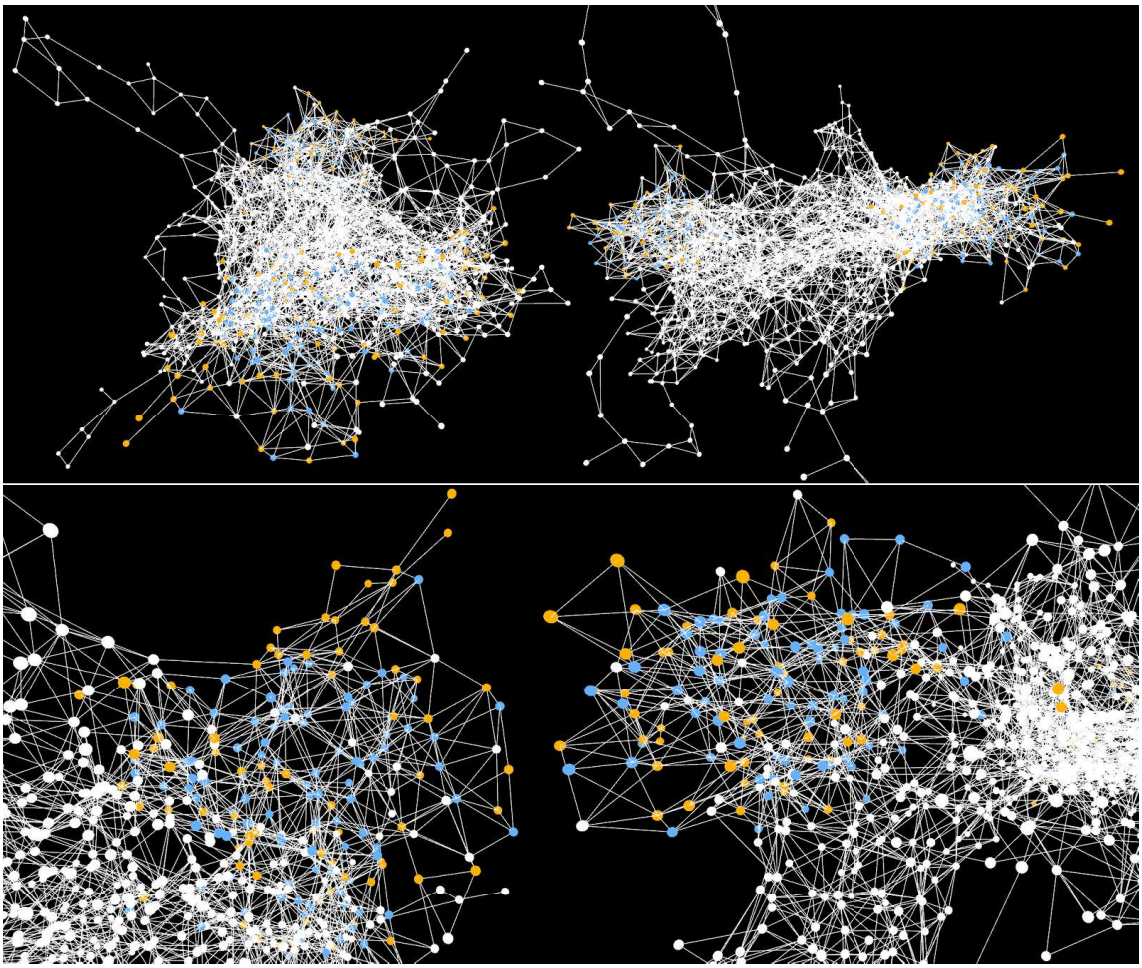

Figure 4.12 Some views of the full 1317-node graph, drawn by the multi-dimensional algorithm, including zoomed-in structures in the areas with more prevalence of labeled nodes.

Removing the unlabeled nodes (and any edges incident on them) yielded an unconnected graph with three connected components. One with 204 nodes and 531 edges, one with 139 nodes and 462 edges, and one with a single node and no edges. The single node component was discarded and the other two components are small enough for the spring algorithm. Since determining which shape they would be prone to take was part of the experiment, the spring algorithm was allowed to run uninterrupted until the nodes stopped moving entirely, which took a couple of minutes, but much of that time was comprised of tiny adjustments after the graphs had already gotten very close to their final shape. The larger component (204 nodes, shown in figure 4.13) needed about 300 iterations to approximate its final layout, while the smaller one (139 nodes, shown in figure 4.14) needed about 200 iterations.

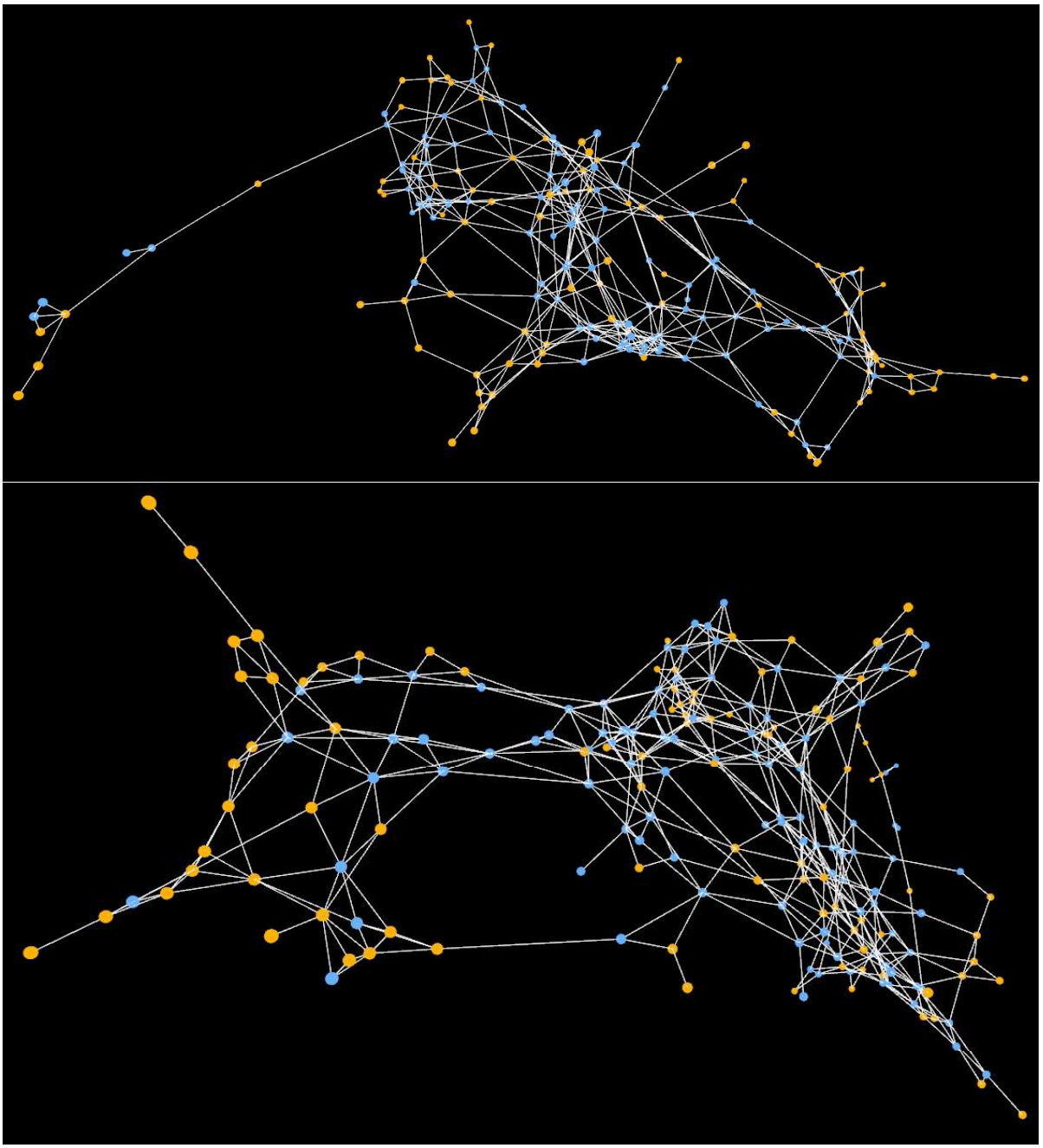

Figure 4.13 Some views of the larger (204-node) component, drawn by the spring algorithm.

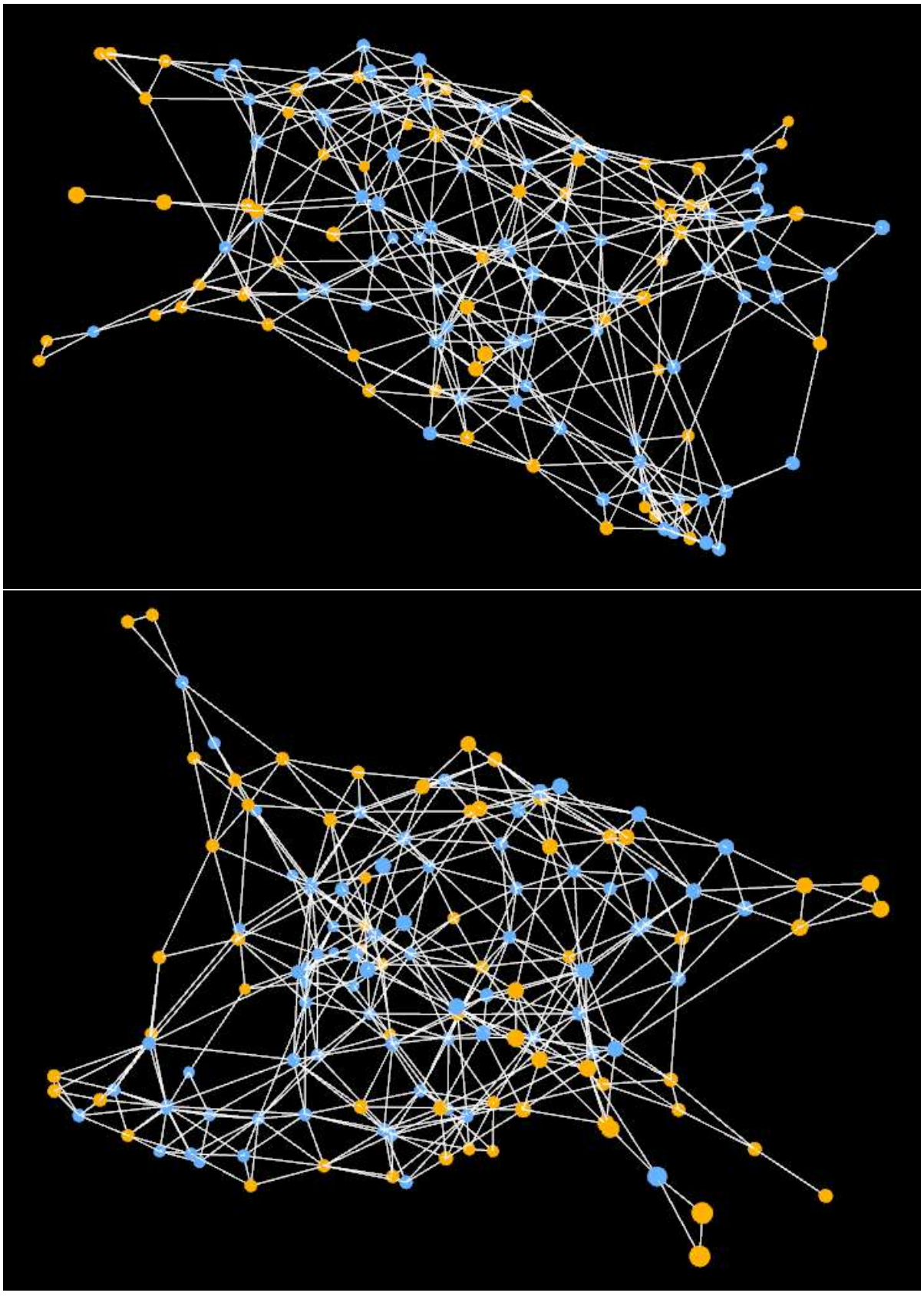

Figure 4.14 Some views of the smaller (139-node) component, drawn by the spring algorithm.

Chapter 5

CONCLUSION AND FUTURE WORK

This proof of concept was able to produce readable and aesthetically pleasing three-dimensional representations for graphs with up to a few hundred nodes. For larger graphs, it was able to produce recognizable shapes, and highlighted which specific issues with the model would require further focus to improve the outcomes.

Future work on this tool mainly encompasses expansion and optimization. As different types of graphs require different approaches to produce readable embeddings (KAMADA; KAWAI, 1989), more drawing options are desirable, especially to be able to properly deal with weighted edges. The adaptation of the refined spring model (FRUCHTERMAN; REINGOLD, 1991) used in section 4.2 showed promising results, and could replace the classic spring method as the primary model for small graphs.

There is room for optimization in a number of aspects. It would be desirable to migrate the bulk of the code away from pure Python; C++ offers better integration with modern OpenGL (which is also a desirable upgrade), faster execution, and more efficient mathematics libraries. That said, the Python interfacing is valuable, especially since this tool is aimed at working with graphs in machine learning (MOONEY, 2022). Cython is an option, as is implementing the algorithms themselves as C++ libraries compatible with Python.

Improvements in visualization itself are also desirable. Utilizing quaternions to guide the camera's focus and orientation granted a lot of freedom, but controlling the camera's position itself when moving orbitally is still done through Euler angles in relation to the global axes, and therefore is still subject to gimbal lock (GOLDMAN, 2011). Extending the quaternion calculations to orbital positioning would solve this issue and also make for more intuitive controls. Other than that, the algorithms described have many parameterizable options, and including all of them in the command-line call would be clunky and unwieldy, so a more sophisticated GUI would be useful.

BIBLIOGRAPHY

ABADI, M. et al. Tensorflow: A system for large-scale machine learning. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. [s.n.], 2016. p. 265–283. Disponível em: ⟨https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf⟩.

ASTURIANO, V. *vasturiano/3D-force-graph: 3D force-directed graph component using Three.js/WebGL*. 2022. Disponível em: ⟨https://github.com/vasturiano/3d-force-graph⟩. Acesso em: 10 oct. 2022.

BERTSIMAS, D.; TSITSIKLIS, J. Simulated annealing. *Statistical Science*, Institute of Mathematical Statistics, v. 8, n. 1, p. 10–15, 1993.

BLENDER FOUNDATION. *blender.org - Home of the Blender project*. 2022. Disponível em: ⟨https://www.blender.org/⟩. Acesso em: 25 nov. 2022.

COFFMAN, E.; GAREY, M. R.; JOHNSON, D. Approximation algorithms for NP-hard problems. In: _____. [S.l.: s.n.], 1996. p. 46–93.

DI BATTISTA, G. et al. *Graph Drawing: Algorithms for the Visualization of Graphs*. [S.l.]: Prentice Hall, Englewood Cliffs, NJ, 1999.

DUJMOVIC, V.; WHITESIDES, S. Three-dimensional drawings. In: _____. *Handbook of Graph Drawing and Visualization*. CRC Press, 2013. Disponível em: ⟨https://cs.brown.edu/people/rtamassi/gdhandbook/⟩. Acesso em: 23 out. 2022.

EADES, P. A heuristic for graph drawing. v. 42, p. 149–160, 1984.

EADES, P.; GARVAN, P. Drawing stressed planar graphs in three dimensions. In: *Proceedings of the 3rd Symposium on Graph Drawing*. [S.l.: s.n.], 1995. p. 212–223.

EVERITT, C. et al. Approaching zero driver overhead. In: GAME DEVELOPERS CONFERENCE, 2014, San Francisco, CA. 2014. Disponível em: ⟨https://www.slideshare.net/CassEveritt/approaching-zero-driver-overhead⟩. Acesso em: 25 nov. 2022.

FRUCHTERMAN, T. M. J.; REINGOLD, E. M. Graph drawing by force-directed placement. *Software: Practice and Experience*, v. 21, n. 11, p. 1129–1164, 1991.

GAJER, P.; GOODRICH, M. T.; KOBOUROV, S. G. A fast multi-dimensional algorithm for drawing large graphs. *Proceedings of the 8th International Symposium on Graph Drawing*, v. 211–221, 01 2000.

GAJER, P.; KOBOUROV, S. G. GRIP: Graph dRawing with Intelligent Placement. In: MARKS, J. (Ed.). *Graph Drawing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. p. 222–228. ISBN 978-3-540-44541-8.

GOLDMAN, R. Understanding quaternions. *Graphical Models*, v. 73, p. 21–49, 03 2011.

GREEN, S. *Evaluating investor performance using Neo4j, GraphXR and ML*. 2019. Disponível em: ⟨https://neo4j.com/blog/evaluating-investor-performance-using-neo4j-graphxr-and-ml/⟩. Acesso em: 20 sep. 2022.

HADANY, R.; HAREL, D. A multi-scale algorithm for drawing graphs nicely. *Discrete Applied Mathematics*, v. 113, n. 1, p. 3–21, 2001.

HAGBERG, A.; SCHULT, D.; SWART, P. *NetworkX Reference*. [S.l.], 2022. Disponível em: ⟨https://networkx.org/documentation/latest/_downloads/networkx_reference.pdf⟩. Acesso em: 25 nov. 2022.

HAMILTON, W. L.; YING, R.; LESKOVEC, J. Representation learning on graphs: Methods and applications. *IEEE Engineering Bulletin*, n. 40, p. 52–74, 2018.

HAREL, D.; KOREN, Y. A fast multi-scale method for drawing large graphs. In: *Proceedings of the 8th International Symposium on Graph Drawing*. Berlin, Heidelberg: Springer-Verlag, 2000. (GD '00), p. 183–196.

HASAL, M.; NOWAKOVA, J.; PLATOS, J. Three-dimensional graph drawing by Kamada-Kawai method with Barzilai-Borwein method. In: *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. [S.l.: s.n.], 2017. p. 1–7.

JONG, N. de. *15 tools for visualizing your Neo4j graph database*. Neo4j Developer Blog, 2021. Disponível em: ⟨https://medium.com/neo4j/15-tools-for-visualizing-your-neo4j-graph-database-ff7315873032⟩. Acesso em: 14 nov. 2022.

KAMADA, T.; KAWAI, S. An algorithm for drawing general undirected graphs. *Information Processing Letters*, v. 31, n. 1, p. 7–15, 1989.

KHRONOS GROUP. *OpenGL® and OpenGL® ES Reference Pages*. [S.l.], 2021. Disponível em: ⟨https://registry.khronos.org/OpenGL-Refpages/⟩. Acesso em: 25 nov. 2022.

KOBOUROV, S. G. Force-directed drawing algorithms. In: _____. *Handbook of Graph Drawing and Visualization*. CRC Press, 2013. Disponível em: ⟨https://cs.brown.edu/people/rtamassi/gdhandbook/⟩. Acesso em: 16 out. 2022.

KUNEGIS, J. *Les Misérables Co-occurrence Network*. 2018. Disponível em: ⟨http://konect.cc/networks/moreno_lesmis/⟩. Acesso em: 15 jun. 2022.

LOPES, T. J. S.; NOGUEIRA, T.; RIOS, R. A machine learning framework predicts the clinical severity of hemophilia b caused by point-mutations. *Frontiers in Bioinformatics*, v. 2, 2022. ISSN 2673-7647. Disponível em: ⟨https://www.frontiersin.org/articles/10.3389/fbinf.2022.912112⟩.

LOPES, T. J. S. et al. Prediction of hemophilia a severity using a small-input machine-learning framework. *npj Systems Biology and Applications*, v. 7, n. 1, p. 22, May 2021. ISSN 2056-7189. Disponível em: ⟨https://doi.org/10.1038/s41540-021-00183-9⟩.

LOPES, T. J. S. et al. Protein residue network analysis reveals fundamental properties of the human coagulation factor viii. *Scientific Reports*, v. 11, n. 1, Jun 2021. ISSN 2045-2322. Disponível em: ⟨https://doi.org/10.1038/s41598-021-92201-3⟩.

MOONEY, P. *Kaggle Survey 2022: All Results*. 2022. Disponível em: ⟨https://kaggle.com/code/paultimothymooney/kaggle-survey-2022-all-results⟩. Acesso em: 25 nov. 2022.

PERUMAL, L. Quaternion and its application in rotation using sets of regions. *International Journal of Engineering and Technology Innovation (IJETI)*, v. 1, p. 35–52, 10 2011.

RICCIO, C. *glm Manual*. [S.l.], 2017. Disponível em: ⟨http://glm.g-truc.net/0.9.8/glm-0.9.8.pdf⟩. Acesso em: 8 nov. 2022.

SYMPY DEVELOPMENT TEAM. *SymPy Development Roadmap*. 2021. Disponível em: ⟨https://www.sympy.org/en/roadmap.html⟩. Acesso em: 30 nov. 2022.

THE GSL TEAM. *Multidimensional Root Finding*. [S.l.], 2021. Disponível em: ⟨https://www.gnu.org/software/gsl/doc/html/multiroots.html⟩. Acesso em: 30 nov. 2022.

THE SCIPY COMMUNITY. *scipy.optimize.fsolve*. [S.l.], 2022. Disponível em: ⟨https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fsolve.html⟩. Acesso em: 1 dec. 2022.

TUTTE, W. T. How to draw a graph. 1963. Disponível em: ⟨https://www.cs.jhu.edu/~misha/Fall07/Papers/Tutte63.pdf⟩. Acesso em: 16 out. 2022.

WHITNEY, H. Non-separable and planar graphs. In: *Proceedings of the National Academy of Sciences of the United States of America*. [S.l.]: National Academy of Sciences, 1932. v. 17, n. 2, p. 125–127.

WU, L. et al. *Graph Neural Networks: Foundations, Frontiers, and Applications*. [S.l.]: Springer Singapore, 2022.

ZHANG, S. et al. Graph convolutional networks: a comprehensive review. *Computational Social Networks 6*, n. 1, p. 1–23, 2019.

ZHOU, J. et al. Graph neural networks: A review of methods and applications. *AI Open*, n. 1, p. 57–81, 2020.

Zuzu-Typ. *Zuzu-Typ/PyGLM: Fast OpenGL Mathematics (GLM) for Python*. 2022. Disponível em: ⟨https://github.com/Zuzu-Typ/PyGLM⟩. Acesso em: 17 oct. 2022.