

Advanced Practical - Building ALVIS, a web-based graph visualization tool

TILLMANN FEHRENBACH, Heidelberg University, Germany

1 INTRODUCTION

We aim to provide an easily accessible visualization tool for graph data and for graph algorithms. This report is an intermediate snapshot of the current state of the project.

In [section 2](#) we want to introduce the reader to the main [tech stack](#) for building such an application. We layout some of the third party libraries used, then give an overview on [WebAssembly](#) and also elaborate on doing [visualizations](#) via WebGL and Three.js. Then we give an overview of the [User Interface](#) and some design choices.

In [section 3](#) we delve into the field of graph drawing. First we introduce to the [general problem](#) and briefly discuss different [graph drawing approaches](#). Then we discuss a [classical Force Directed](#) graph drawing algorithm and an improvement to [scale the approach](#) to larger graph data sets using Multidimensional Scaling and Stochastic Gradient Descent. Additionally we look at two other approaches to [visualize tree data sets](#) i.e. spherical tree drawing in Euclidean space and on the Poincare Disk.

In [section 4](#) we provide a brief conclusion and ideas for future work.

2 WEB DEVELOPMENT OVERVIEW

2.1 Tech Stack

A tech stack is a combination of programming languages, databases, and frameworks used to create a complex web application or website. A typical web development stack is usually a mix of front-end and back-end technologies that include:

- **Front-end Framework:** Libraries of code written by other developers. These can help building a web application without starting from scratch. We use React.js, a widely used framework for building user interfaces. The architecture of React.js is based on the concept of virtual DOM (Document Object Model). React.js keeps track of changes in the virtual DOM and only updates the parts of the actual DOM that need to change which is great for user interactivity. It works by breaking down a user interface into components and rendering these components on the page as HTML.
- **Front-end Back-end communication:** Since we use Node.js as a JavaScript runtime, a natural way to handle HTTP requests and responses and to do URL routing was Express.js. The communication between the front-end and back-end should be reduced as much as possible in this project. Hence Express is merely used to send new data sets from the back-ends database to the users browser, which makes the design of a complicated back-end infrastructure redundant and tries to use the users hardware resources as best as possible.
- **Databases:** The user should be able to choose from a wide variety of graph data sets or store computed visualizations of those externally. Any data is stored as a JSON file in a MongoDB database and it can be transferred to and from the front-end. We chose MongoDB due to its ease of use and its perfected integration into Express/Node.js. The need for a complicated relational data base seemed an overkill when designing the general architecture of the app. Going forward, these design choices might change.

- Programming languages: For every problem to be solved there is a whole set of programming languages available. JavaScript has been the main language for bringing logic to a website for the last 20 years, while HTML brings structure to a web-page and CSS makes things look "nice". Even though this is still common practice nowadays, there are issues with JavaScript, mainly with performance, since it is an inherently interpreted language. It works fine for a standard web applications where small pieces of processing is done inside the web browser, based on user interaction. Generally, the heavy processing is done on servers, and the browser interacts with them through http services. But this transfer is an important bottleneck when lot of real-time processing is required, for example image video processing, 3D gaming, AR/VR and just complex algorithmic problems in general including algorithms that act on huge graph data sets. One common approach nowadays is to outsource the required computations via WebAssembly on the users device directly.

2.2 An introduction to WebAssembly

Web Assembly (Wasm) is a low-level binary format for executing code on the web. It is designed to be a fast and efficient way of delivering applications and high-performance code to the web browser, with performance close to native code. It is supposed to work alongside JavaScript, allowing developers to write parts of their application in another low level language such as C++, resulting in a hybrid application as it is in our case. Wasm itself is a compilation target, which means that code written in another programming language can be compiled to Wasm and run in a web browser via a specialized compiler, i.e. the Emscripten compiler for C/C++. It is designed to be secure and sand-boxed, meaning that code executed in Wasm runs in a secure environment that is separate from the rest of the web page.

Wasm also provides a number of advanced features, including low-level memory access and multi-threading support. In the case of C++ we can also make use of third party libraries such as Boost for very optimized linear algebra operations. Some libraries need to be passed to the compilation process as well, making the compilation process rather slow, others (e.g Boost) have already been precompiled by others to the appropriate format and can be included during the compilation process efficiently. The main drawback of using WebAssembly nowadays is the lack of a big community. The documentation for integrating WebAssembly into a React App is basically non existent, which made the development very tedious.

Our design choice was to outsource all computationally expensive/interesting parts to a C++ Wasm Module. This includes finding the proper Layout of the data set (more on that in the section about Graph drawing) and executing an algorithm on a given graph data set by using our Visualization Logger (more on that in the Data Flow part). Figure 1 gives an overview of the general workflow of a Wasm component. The important part is that the WebAssembly can run in a separate web-worker, which prevents blocking the main thread (UI).

2.3 Handling graphics in the Browser

The goal of our App is to provide interactive 2D and 3D Visualizations of Data, hence the choice of the right Graphics Pipeline was an important one. WebGL (Web Graphics Library) is a low-level JavaScript API for rendering interactive 3D graphics within a web browser, by communicating directly with the GPU/integrated Graphics Cards of the user's device, allowing for high-performance graphics rendering. WebGL is a wrapper around the relatively old but still popular OpenGL. Three.js is a widely used abstraction layer on top of WebGL, that lets the user focus on the manipulation of a geometry in space over time, rather than dealing with the rendering process, which includes writing vertex shaders, the rasterization process and applying colors and textures via fragment shaders. We make use of the primitive geometries provided by Three.js in order to display a

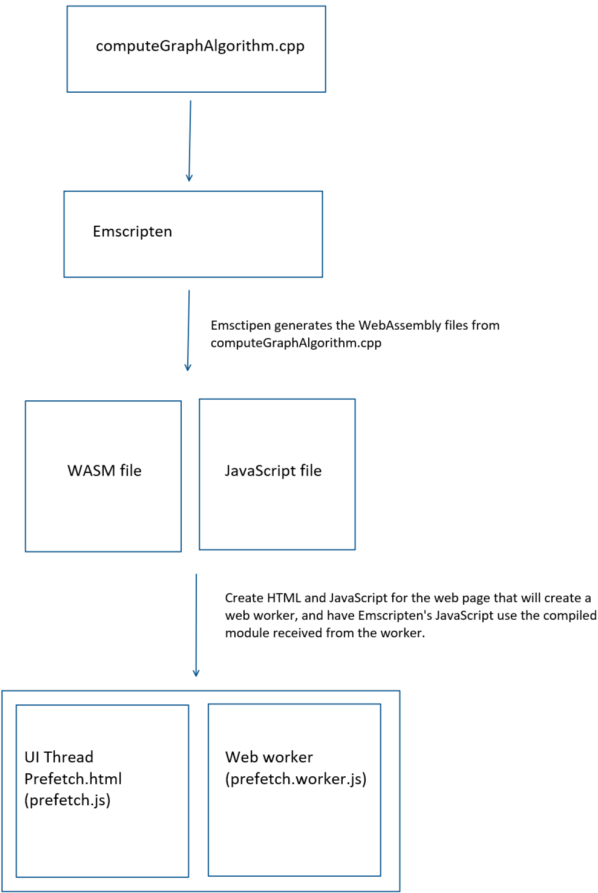


Fig. 1. General Workflow of a Wasm component.

graphs nodes (Spheres) and edges (Bezier-Splines). Morphing geometries over time requires an iterative update of all the vertices of a given mesh, which becomes a bottleneck for huge graphs. Since our visualization consists of repeating geometries, 'instancing' comes in handy, which is the practice of rendering multiple copies of the same mesh in a scene at once, by duplicating vertex data across all instanced meshes. Instancing still allows partial differentiation of attributes, such as position and color.

At the time writing this article, there have been some major design changes, which include a more modern approach of GPU programming, namely WebGPU. WebGPU is considered to be the future standard for Web Browsers to do all sorts of Graphics. It is a GPU API written in the Rust Programming Language (very fast), that serves as a wrapper around the newer Graphics API's such as Vulkan etc. Most importantly, it is possible to write Compute Shaders in WebGPU which makes General Purpose Programming on the GPU possible. The shading language is not GLSL (OpenGL) but WGSL, a shading language with a Rust-like syntax. WebGPU will slowly but steadily replace large parts of our apps WebAssembly part, since it allows highly parallelized computation which is

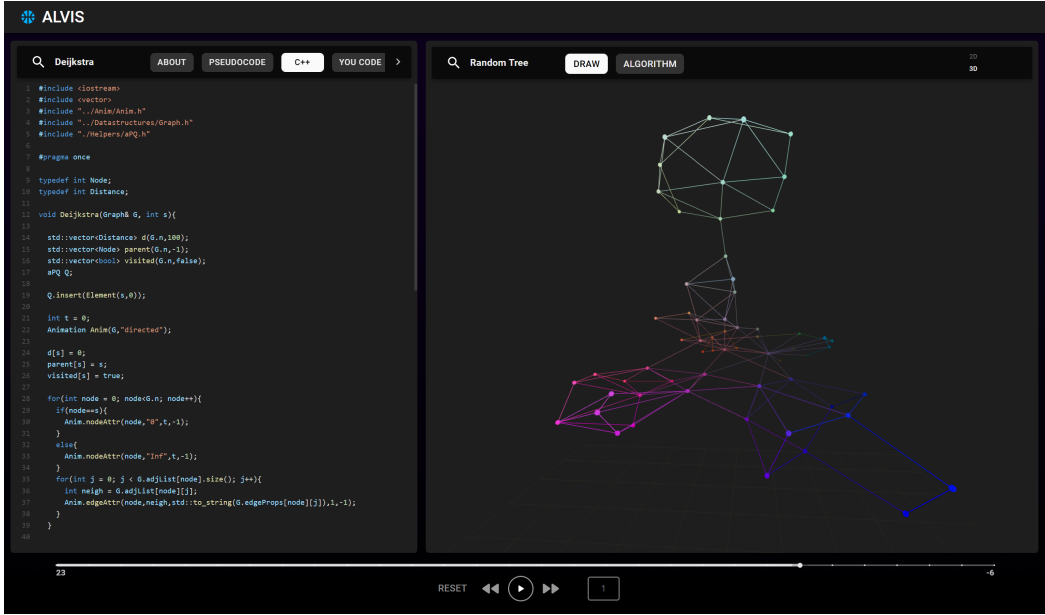


Fig. 2. The User Interface

of great advantage, especially when it comes to SIMD tasks such as large linear algebra operations. A future report will elaborate on that in much more depth.

2.4 Designing the UI

Our goal was to design a user friendly interface that enables the user to choose what data set shall be displayed and what type of algorithm shall be visualized. The UI can be seen in Figure 2. On the left side of the screen, the user is able to pick between an explanatory (About) section written with Mathjax, that provides a more or less interactive introduction to the general problem that some chosen graph algorithm tries to solve, furthermore a section with plain Pseudocode and the actual implementation in C++ can be seen. The C++ implementation section highlights active lines of the code during an algorithms visualization. On the right hand side the actual graph is being displayed. The user can chose between the Algorithms Visualization (default) but also display the evolution of the graph drawing over time, usually a mesmerizing process in which a cloud of randomly positioned points unfolds to a beautiful geometric structure. The user is also able to pick between a visualization in 2D and 3D. On the bottom of the page a time line shows the progress of an algorithm and lets the user freely jump forth and back in time. The speed of the animation can also be adjusted. In the search bar on the left an algorithm from a predefined set of algorithms can be picked (all Wasm files), in the search bar on the right the data set from a predefined set of graph data sets can be chosen which is then served by the MongoDB database.

Since the computationally expensive WebAssembly Parts are not done on the main thread, the general user experience is smooth and enables true interactivity, which was after all the main goal of this project.

2.5 Data Flow and Logging State

The state changes that yield the later algorithms visualization are calculated during the actual algorithms execution inside a web-worker. We wrote a simple C++ logger library for graph algorithms that enables the designer of a visualization to pick a graph algorithm and mark active nodes or edges during the execution. The library will automatically log the events without further interference. The designer (not the user of the app) can hence pick an existing algorithm and easily expand it with the logging commands. For example when trying to visualize a network flow algorithm, the active and non active nodes and edges can be colored differently, but also the values (such as the excess of a node or the flow through and edge) can be logged. During the algorithms visualization in the UI, all state changes are considered at every time step. An important note is that the logger only logs the actual changes of states and not the full state of the graph at any given time in order to minimize resources and the keep the JSON file that gets transferred from the C++/Wasm part to the JavaScript/WebGl part as small as possible. Figure 3 gives a more detailed overview of the data flow between the JavaScript and WebAssembly components.

3 VISUALIZING GRAPH DATA

3.1 What is a Graph?

A graph $G = (V, E)$ is a mathematical representation of a set of objects and their relationships. The nodes V represent the objects, while the edges E represent the relationships between these objects. Graphs can be used to model many real-world situations, such as social networks, transportation systems or communication networks. Graph algorithms are used to solve a wide range of problems, including path finding, maximum Flow, minimum spanning trees, graph coloring etc. There are several types of graphs that can be visualized with our app, including directed graphs (where edges have a direction), undirected graphs (where edges have no direction) or weighted graphs (where edges have a weight or cost).

3.2 What is graph drawing?

Graph drawing is an area in mathematics and computer science combining methods from geometric graph theory and information visualization to derive two-dimensional or three-dimensional depictions of graphs. The goal of graph drawing is to provide a meaningful and visually appealing representation of a graph, allowing for easy interpretation and analysis of the relationships between elements.

There are a lot of different quality measures defined for graph drawing, that are crucial for aesthetics and usability such as the number of edges that cross each other (crossing number), preservation of symmetry and finding these symmetry groups, the area and aspect ratio and mostly to keep the edges between nodes as straight and short as possible.

There are several different approaches to meet these requirements. A few of them include:

- Force-directed graph drawing: This approach uses physical simulations, such as forces between nodes, to determine the layout of a graph.
- Tree drawing: This approach focuses on visualizing hierarchical structures, such as trees and directed acyclic graphs. The goal is to create a clear and concise representation of the parent-child relationships in the graph.
- Circular layout: This approach arranges nodes in a circular arrangement, choosing carefully the ordering of the vertices around the circle to reduce crossings.

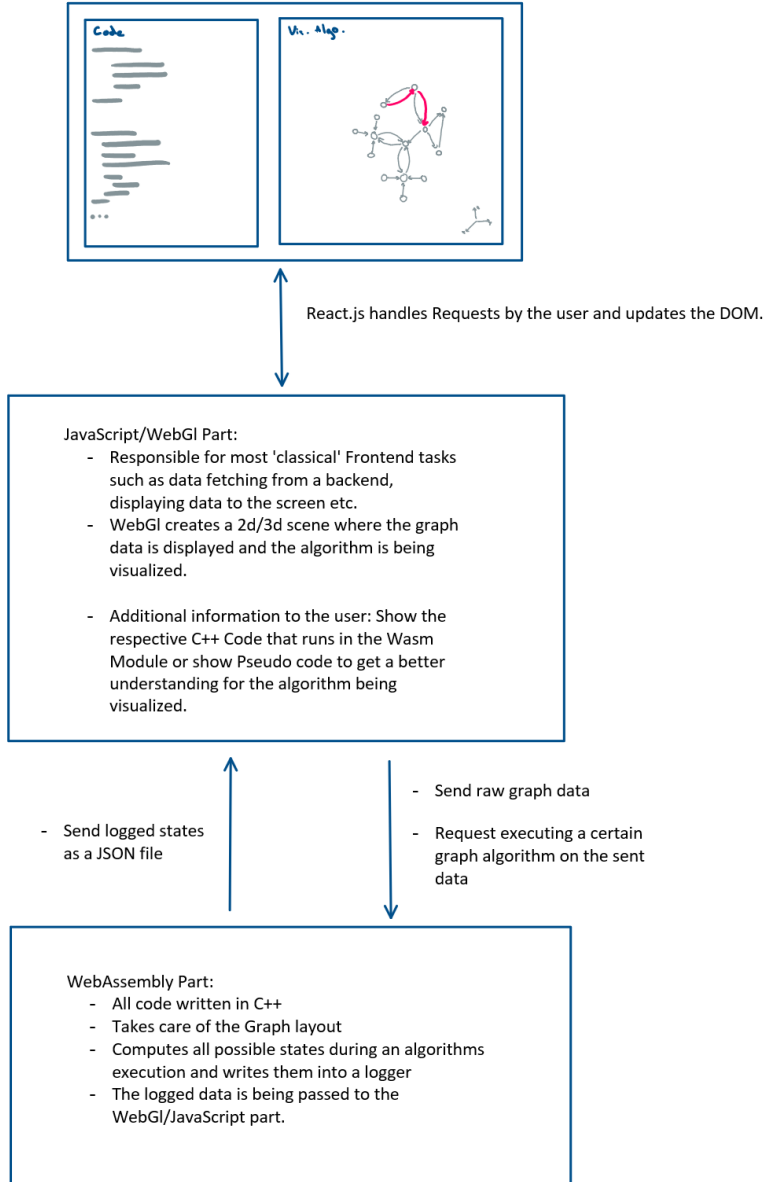


Fig. 3. Overview of the data flow between the components.

- Arc diagrams: This approach places vertices on a line, the edges are drawn as semicircles above or below the line, or as smooth curves linked together from multiple semicircles.

The choice of approach depends on the specific needs and requirements of the graph being drawn. In general the above approaches, such as force directed-and tree drawings, are not bounded to euclidean space. For example, a drawing on the Poincare Disk might be preferred to preserve some hyperbolic properties of a graph or to simply have a fish-eye view on a part of the graph. In

the following we want to look at the most common approach, the force directed graph drawing, and an approach to scale the algorithm to very large graphs efficiently.

3.3 In depth: force directed Graph drawing

Force-directed graph drawing is a graph drawing method by simulating physical forces between nodes in a graph. Usually, spring-like forces based on Hooke's law are used to attract pairs of nodes of the graph, simulating the nodes as points of mass in the underlying space. A common simplification is to only compute attractive forces between nodes that are actually connected through an edge in the graph. Simultaneously, repulsive forces based on Coulomb's law are used to separate the nodes at the same time, acting as if the nodes were similarly charged particles. Even though this approach is inspired by nature, the underlying forces do not necessarily need to follow the actual physical behaviour, e.g. attractive forces might not be linear. Usually an optimal distance between two nodes is defined based on some heuristic, for example Fruchterman and Reingold [2] define the optimal distance between vertices as

$$k = C \sqrt{\frac{\text{area}}{\text{number of vertices}}} \quad (1)$$

(*area* is the space in which the graph can be drawn in, C is an experimentally found parameter which the authors do not specify in greater detail) and then compute the attractive and repulsive forces as

$$f_a(d) = \frac{d^2}{C} \text{ and } f_r(d) = \frac{-k^2}{d} \quad (2)$$

where d is the the euclidean distance between two nodes. A prior approach by Eades [1] used logarithmic attractive forces (expensive). There are three steps to each iteration of the algorithm: Calculate the effect of attractive forces on each vertex; then the effect of repulsive forces; and finally limit the total displacement by the temperature, another empirically chosen parameter that reduces the displacement over time. Even though this approach works quite well for small to medium graphs, speed ups are very complicated and more importantly, only taking the euclidean distance between pairing nodes into account for the the computation of the attractive forces, is a big limitation. Even though Fruchtermann's algorithm got replaced by the approach in the following subsection, it was a helpful and necessary starting point for our graph visualization.

3.4 In depth: force directed Graph drawing via SGD

Taking the distance in euclidean space as the optimal distance between nodes, generally leads to some unsolvable inaccuracies in the drawing. Multidimensional scaling (MDS) is a technique to solve this type of problem. It attempts to minimize the disparity between ideal and low-dimensional distances. Distance scaling is most commonly used for graph drawings. We can write out an

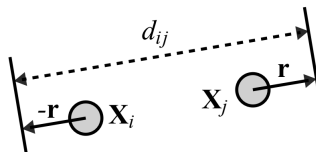


Fig. 4. Pairwise distance update between two randomly selected nodes, following the optimal distance constraint. (Image from [3])

objective that we want to minimize, also called the error function or the stress in the system.

$$\text{stress}(X) = \sum_{ij} \omega_{ij} (\|X_i - X_j\| - d_{ij})^2 \quad (3)$$

where X contains the coordinates of each vertex in low-dimensional space, and d is the ideal distance between them. A weighting factor ω is used to either emphasize or dampen the importance of certain pairs. For the problem of graph layout, the most common approach is to set d_{ij} to the shortest graph theoretic path distance between vertices i and j , with $w_{ij} = d_{ij}^{-2}$ to offset the extra weight given to longer paths due to squaring the difference. Zheng et al. [3] from 2018 describe a method of minimizing stress by using stochastic gradient descent (SGD), which approximates the gradient of a sum of functions using the gradient of its individual terms. In this approach this corresponds to moving a single pair of vertices at a time, i.e. in each iteration a pair of vertices in the graph is chosen at random. Pseudocode of the algorithm is given in Algorithm 1.

Finding All Pairs Shortest Paths (APSP) in a given graph is an overhead in the algorithm. In order to reduce the the amount of paths computed, we use randomly placed pivot nodes whose shortest paths are used as an approximation for the shortest paths of vertices close to them. Every non-pivot node in the graph computes and stores its closest Pivot. The actual distances between all pairs of pivots are being computed before the actual drawing algorithm via bidirectional Dijkstra. If two nodes have the same pivot node, the actual distance between them is computed in order to increase the resolution and prioritize close nodes, having in mind that close nodes affect the local layout more than nodes that are far away from each other. It is important to note that the amount of pivots used in a graph can greatly affect the later layout of the graph, we generally chose 1 to 5 percent of nodes as pivots for larger graphs (above 1000 nodes).

```

input : graph  $G = (V, E)$ 
output:  $k$ -dimensional layout  $X$  with  $n$  vertices
 $d_{\{i,j\}} \leftarrow$  shortest Paths
 $X \leftarrow \text{randomMatrix}(n, k)$ 
for  $\eta$  in annealing schedule do
  foreach  $\{i, j : i < j\}$  in random order do
     $\mu \leftarrow \omega_{ij} * \eta$ 
    if  $\mu > 1$  then
       $\mu \leftarrow 1$ 
    end
     $r \leftarrow \frac{\|X_i - X_j\| - d_{ij}}{2} \frac{X_i - X_j}{\|X_i - X_j\|}$ 
     $X_i \leftarrow X_i - \mu r$ 
     $X_j \leftarrow X_j + \mu r$ 
  end
end

```

Algorithm 1: Stress minimization via Stochastic Gradient Descent as described in Zheng et al. η is a step size that tends towards 0 as the iteration number increases, r is the direction of displacement that the authors derive from the gradient of the stress function. Figure 4 provides a visualization of the mentioned forces.



Fig. 5. Showcase of a converging layout using the the stress minimization algorithm from Algorithm 1 with the distance approximation using 5 pivot nodes.

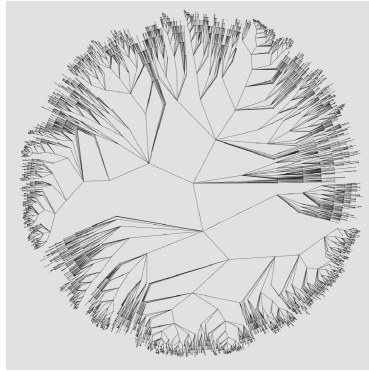


Fig. 6. A simple tree (ca. 4000 nodes) drawing in euclidean space using recursive angle distribution based on the weight of a nodes sub-tree.

3.5 Tree drawings and layouts on the Poincare Disk

During the development of the app we also tried out other drawing approaches. The first one was a simple spherical [tree drawing](#). A very common approach is to recursively allocate an angle of a circle (with radius equal to the current depth of the tree node), depending on the relative weight of its sub-tree compared to the weights of the sub-trees of its sibling nodes. The resulting drawing is nicely balanced, shows the hierarchical structure of the data set and can be computed fast (linear time in the amount of nodes). Figure 4 shows such a drawing, the distance to the root is exponentially decreasing, which is a cosmetic operation after the actual algorithm finished. A simple modification transfers this approach to a drawing on the [Poincare disk](#), where the distances between two nodes p and q is not measured with the euclidean norm but with the following formula:

$$d(p, q) = \operatorname{arcosh} \left(1 + 2 \frac{\|p - q\|^2 r^2}{(r^2 - \|p\|^2)(r^2 - \|q\|^2)} \right) \quad (4)$$

4 CONCLUSION AND FUTURE WORK

Even though the overall User Interface is functional and working, there are still a lot of things that need further improvement, such as increasing the expressiveness of the C++ logger, in order to be able to visualize coarsening or partitioning of a graph. The most important part for now would be to increase the amount of algorithms to choose from. This requires a lot of work since not only

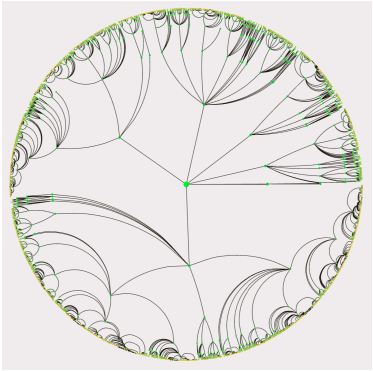


Fig. 7. Tree Layout using Distances as described in Equation (2).

does every algorithm need to be designed individually with our logger, but also the rest of the user interface such as the interactive introductory part in Mathjax has to be tailored to each problem. We believe that our layout of combining a modern Framework (Next.js/React.js) with WebAssembly is well suited for future projects. In order to increase performance, the use of general purpose programming on the GPU (GPGPU) via WebGPU will take a larger part in the future of this project, since it enables highly parallel execution of algorithms. One possible use case is to expand our visualization framework to the field of Deep Learning algorithms, due to their inherent graphical structure.

REFERENCES

[1] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[2] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991. doi: <https://doi.org/10.1002/spe.4380211102>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380211102>.

[3] Jonathan X. Zheng, Samraat Pawar, and Dan F. M. Goodman. Graph drawing by stochastic gradient descent, 2017. URL <https://arxiv.org/abs/1710.04626>.