Creating Multi-Level Visualizations for Large Network Data

Kathryn Gray

Spring 2022

Contents

1 Abstract

In our current data environment, we often have large, network datasets. These can arise from many applications including social networks, science networks, ontology networks, and many others. Additionally, visualizing these datasets can be very important to understand how the networks are connected, the span of data in the network, and where unexpected or unusual things in the network could occur.

We create a multi-layer visualization that allows users to understand the large networks quickly. These multi-layer visualizations rely on extracting the layers from an initial, large graph. We do this using Steiner Trees.

Our first paper consists of the extraction and layout algorithms. This allows us to create a layout that is crossing-free (in edges) and overlap-free (in labels). We also consider other graph layout aesthetics, compactness and desired edge-length realization. Finally, since we are looking at large networks, we include a method that scales well to large datasets.

The next paper, still in progress, works on the visualization system to create a good user experience with these layouts. Here we create the system that allows users to view graphs laid out in this way. We include many user interactions including zooming in, panning, and search. We are working on including other user interactions such as path finding and moving nodes to new layers.

Finally, since we have this visualization system, we propose to perform user studies to show how our system compares to previous systems.

2 Introduction

Exploring large networks is a problem that arises in many fields. Currently, there are only a few methods to even show large datasets and none of these allow the users a comprehensive way to navigate and explore the data.

We build a system based on map data. Since people have been using maps, and especially online maps such as google maps for a long time, a lot of the interactions of maps are known to even non-experts. People use google maps to locate places, discover paths between them, and understand where they are in the world. We create a system that uses these same interactions to provide a starting point to understand large, network data.

To create these map-like systems, we start by defining levels. These levels are analagous to defining towns, cities, and villages. When you are at a low level of zoom, e.g. you can see several states on a map, only the cities or most important nodes should be shown. As you zoom closer, you can start to see the lower level nodes, or towns and villages.

The next step is ensuring all of these nodes are connected at every level. We do this using Steiner trees. This gives us a tree of connected nodes whenever anyone zooms into the graph. This also helps us define important edges, or give edges differences, such as highways and streets. This definition will give us desired edge lengths, a metric we use to lay out the graph.

After we have defined these, we can create the underlying map. Here we take the base graph and create a layout that contains no edge crossings or label overlaps. This algorithm is based on a force directed algorithm, where we disallow crossings and remove most overlaps. A post-processing step ensures no overlaps. We have an option for this algorithm to allow batch processing which can speed up the process. Once the tree has been laid out, we use a clustering method and a method to create the “countries".

At this point, we are ready to create an interactive experience in the browser for people to see their data. We allow users several interactions with the graph and are creating a few more. Users are able to pan, zoom, and search. A location tracker in the corner allows users to see where there are in the map.

Finally, we are looking to test whether this method will actually allow users to understand the large data better. We are working with some collaborators at Sandia National Laboratories that have some large graph data they would like to understand better. We plan to work with them to create a case study of the system. Additionally, we are looking to understand how the general public can understand network data better, we will create a user study to look at how our system works and find any improvements from that.

3 Previous Work

Tree and Network Layout Algorithms: Drawing trees has a rich history: Treevis.net [38] contains over three hundred different types of visualizations. Here we briefly review algorithms for 2D node-link representations, starting with arguably the best known one by Reingold and Tilford[37]. This and other early variants draw trees recursively in a bottom-up sweep. These methods produce crossings-free layouts, but do not consider node labels or edge lengths. We will now broaden our scope to general graphs. While general graphs are not necessarily planar, the layout techniques and ideas can be applied to trees. Most general network layout algorithms use a force-directed [13, 15] or stress model [8, 27] and provide a single static drawing. The force-directed model works well for small networks, but does not scale to large networks. Speedup techniques employ a multi-scale variant [21, 16] or use parallel and distributed computing architecture such as VxOrd [7], BatchLayout [36], and MULTI-GILA [3]. Libraries such as GraphViz [14] and OGDF [11] provide many general network layouts, but may not support interactions. Whereas visualization toolkits such as Gephi [5] and yEd [41] support visual network manipulation, and while they can handle large networks, the amount of information rendered statically on the screen makes the visualization difficult to use for large networks.

Overlap Removal and Topology Preservation: In theory nodes can be treated as points, but in practice nodes are labeled and these labels must be shown in the layout [34, 22]. Overlapping labels pose a major problem for most layout generation algorithms, and severely affect the usability of the resulting visualizations. A simple solution to remove overlaps is to scale the drawing until the labels no longer overlap. This approach is straightforward, although it may result in an exponential increase in the drawing area. Marriott *et al.* [30] proposed to scale the layout using different scaling factors for the x and y coordinates. This reduces the overall blowup in size but may result in poor aspect ratio. Gansner and North [19], Gansner and Hu [18], and Nachmanson *et al.* [32] describe overlap-removal techniques with better aspect ratio and modest additional area. However, these approaches can and do introduce edge-crossings, even when starting with a crossings-free input. Placing labeled nodes without overlaps has also been studied [29, 31, 24], but these approaches also cannot not guarantee crossings-free layouts.

The need for new algorithms: While there exist many algorithms for generating tree and network layouts, to the best of our knowledge, no existing algorithm considers the four aspects of the readability of labeled tree layouts: no edge crossings, no node overlaps, compact drawing area and preserved desired edge lengths. For example, one of the most frequently used network visualization systems, GraphViz [14], has an efficient layout algorithm based on the scalable force directed placement (sfdp) algorithm [23] and can remove label overlaps via the PRoxImity Stress Model (PRISM) [18]. The output, however, does not optimize the given edge lengths and cannot ensure the crossing constraint; examples in this paper contain 100-1000 crossings.

The popular visualization library d3.js provides a link-force feature to optimize desired edge lengths, but cannot ensure the crossing constraint and cannot remove label overlaps without blowing up the drawing area. Another excellent visualization toolkit, yEd [41], provides a method that can draw trees without edge crossings and optimize compactness. However, none of the methods available in yED can preserve the desired edge lengths and one cannot remove label overlap without blowing up the drawing area. Nguyen and Huang [33] describe an algorithm for compact tree layouts (note that we use a similar initialization step), however, their approach is not concerned with edge lengths and node labels.

Different dimensionality reduction techniques such as t-SNE [40] and its variants [25, 28] are hard to use to visualize tree networks since they do not guarantee crossing-free and label overlap-free drawing;in one of our experiments, we observed that t-SNE can generate more than 50 crossings in a small tree of 100 nodes. We believe our paper fills this gap in the literature by developing and making available a scalable method for readable tree layouts.

4 Current Contributions: Creating a Layout and Initial Visualization

4.1 Extracting Layers, Multi-Level Steiner Tree

A Steiner tree minimizes the total weight of the subtree spanning a given subset of nodes (called the terminals). The multi-level Steiner tree problem is a generalization of the Steiner tree problem where the objective is to minimize the sum of the edge weights at all levels. As both problems are NP-hard, we use approximation algorithms that have been shown to work well in practice [1].

Formally, given a node-weighted and edge-weighted network $G = (V, E)$, we want to visualize G with the aid of a hierarchy of progressively larger trees $T_1 = (V_1, E_1) \subset T_2 = (V_2, E_2) \subset \cdots \subset T_n = (V_n, E_n) \subseteq G$, such that $V_1 \subset V_2 \subset \cdots \subset V_n = V$ and $E_1 \subset E_2 \subset \cdots \subset E_n \subset E$. We use multi-level Steiner trees in order to make the hierarchy representative of the underlying network, based on a node filtration $V_1 \subset V_2 \subset \cdots \subset V_n = V$ with the most important nodes (highest weight) in V_1, the next most important nodes added to form V_2, and so on. A solution to the multi-level Steiner tree problem then creates the set of progressively larger trees $T_1 = (V_1, E_1) \subset T_2 = (V_2, E_2) \subset \cdots \subset T_n = (V_n, E_n) \subseteq G$ using the most important (highest weight) edges.

4.2 Readable Tree Layout Algorithm

Our algorithm can be split into five parts, multi-level Steiner tree extraction, an initial layout, a force-directed layout improvement, a final iteration ensuring no label overlaps, and a map-like visualization; see Fig. **??**. Since we rely on existing techniques for the first step and the last steps, we briefly discuss them in Sec. 4.5. In this section, we discuss the remaining three steps in detail.

4.2.1 Initialization

mentioned in Sec. **??**, desired edge length preservation and compactness are contradictory optimization goals. Hence, our algorithm can produce two different types of layouts: one emphasizing the desired edge length preservation and the other emphasizing compactness. The two of intialization steps below influences the type of the obtained layout.

Edge Length Initialization:

The first initialization creates a layout that is crossing free and preserves all edge lengths, although it may have label overlaps. Our crossings-free initialization is similar to a prior work [4]. We select a root node and assign a wedge region (sector) to every child. Each child is then placed along an angle bisector of the assigned wedge, away from its parent by the desired edge length. We continue this process until the coordinates for each node have been computed; see pseudcode Alg. `Edge-Length-Initialization`. We traverse the nodes using breadth-first search (BFS) which visits parents before children. Since the angular regions are unbounded, the algorithm can preserve all the desired edge lengths exactly. When we assign wedge regions to child nodes, the angles are proportional to the size of the subtree rooted from the child; see Fig. 2. When the input is a balanced tree this algorithm computes a symmetric layout; see Fig. 1.

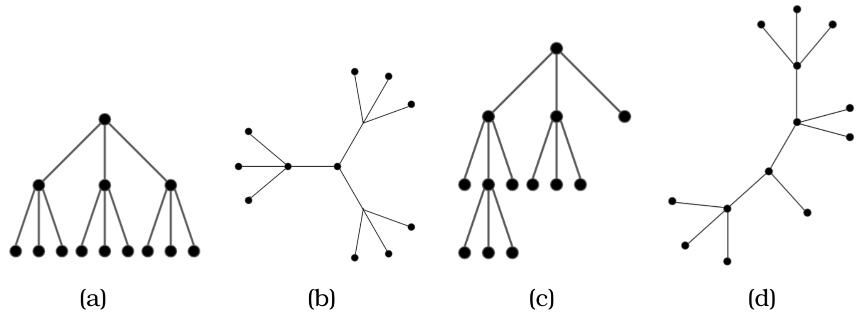


Figure 1: Illustrating the output layout of `Edge-Length-Initialization` w.r.t. different types of inputs. If the tree is balanced (Fig: 1a) then the output is symmetric (Fig: 1b). If the tree is not balanced (Fig: 1c) then the output is not symmetric (Fig: 1d).

Compact Initialization:

The second initialization also creates a crossing-free layout that is compact, although desired edge lengths might be ditorted. Instead of assigning wedge regions to children of a node, we assign the entire fan area to the children and place children along the arc of the fan; see pseudcode Alg. `Compact-Initialization`. This results in a wider spread of nodes, but does not preserve the desired edge lengths; see Fig. 3 and compare it with Fig. 2.

Algorithm 1: `Edge-Length-Initialization`

```
Input: $G = (V, E)$ // The tree network
$root \in V$ // root of the tree
$\{\dots, n_v, \dots\}$ // Size of subtree rooted from $v \in V$
$\{\dots, DEL_{uv}, \dots\}$ // Desired edge length from $u$ to $v \in V$
Output: $X$ // Crossing-free initial layout for RT_L
Function Initial_Layout($G, root$):
    $X_{root} \leftarrow (0, 0)$;
    $W_{root} \leftarrow wedge(center = root,$
        $radius = DEL_{root}, angle\_range = [0, 2\pi))$;
    for parent node $p \in BFS(G, root)$ do
        $\{c_1, c_2 \dots c_{|C|}\} \leftarrow children(p)$;
        $angle\_ranges = \{\dots A_{c_i} \dots\} \leftarrow$
          $partition(angle\_range(W_p); n_{c_1}, n_{c_2}, \dots)$;
        for $c_i \in C$ do
            $W_{c_i} \leftarrow wedge(center = p,$
                $radius = DEL_{c_i}, angle\_range = A_{c_i})$;
            $X_{c_i} \leftarrow midpoint(arc(W_{c_i}))$;
    return $X$;
```

4.2.2 Force Directed Improvements

The next step is to use a force directed algorithm to optimize our soft constraints. Here we describe all of the forces used, as well as how we prevent edge crossings at every step of the force directed improvement.

4.2.3 Maintaining the Crossings-Free Constraint

The force-directed algorithm improves the layout by applying different forces while ensuring that there are no edge crossings introduced in any iteration of the algorithm. The algorithm starts with a layout computed in the previous initialization step. In each iteration of the algorithm, it computes different forces for each node as discussed in the next sections. Then for each node v, it computes the movement T_v applied by the forces. The algorithm combines the forces linearly, with scaling factors discussed in Sec. 4.4. If T_v introduces any edge crossings, then the algorithm does not apply the movement. Otherwise, it computes the new coordinate of v according to T_v. The algorithm continues this step until a maximum number of iterations is reached; see pseudocode Alg. `Force-Directed-Improvement`.

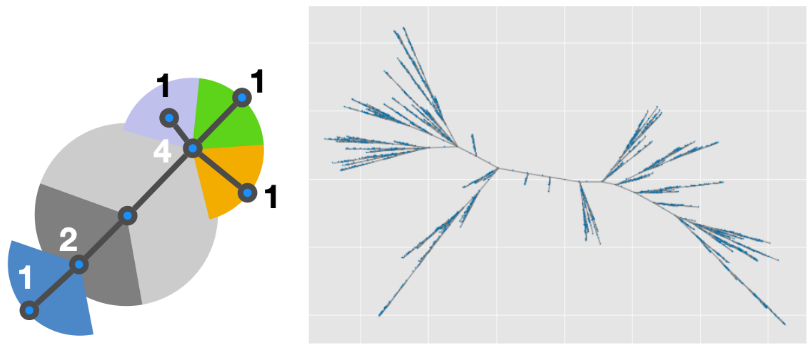


Figure 2: **Left:** Illustration of initial radial layout of RT_L. The numbers indicate the proportion of wedge sectors determined by the size of induced subtrees of child nodes. **Right:** Initial layout of the last.FM network.

4.2.4 Label Overlap Force

We use an elliptical force by modifying the traditional collision force to help remove the label overlaps. Since labels are typically wider than they are tall, a circular collision region potentially wastes space above and below the labels. We build an elliptical force out of a circular collision force by stretching the y-coordinate by a constant factor b (e.g., by default we use $b = 3$) before a circular collision force is applied, and restoring the coordinates after the force is applied. The velocity computed by the collision force is processed in a similar manner, with a reciprocal scaling factor. Formally, let X_v denote the coordinate of node v. We specify a different collision radius depending on label size, denoted by r_v, for every node v. Note that the collision radius depends on both the font size of a label and the number of characters in the label. A circular collision force first calculates a movement T'_v, and then the elliptical movement T_v is computed by stretching the x-coordinate and compressing the y-coordinate, e.g., $T_v.x = T'_v.x \times b, T_v.y = T'_v/y$. Then we update the coordinate X_v by adding T_v.

4.2.5 Edge Length Force

The edge length force is designed to maintain the desired edge lengths. For every edge we apply either a repulsive force $f_r^e = K/d \cdot I_{d<l_e}(d)$ (when the edge is compressed) or an attractive force $f_a^e = Kd \cdot I_{d>l_e}(d)$ (when the edge is stretched), determined by the indicator function. The force is proportional/reciprocal to distance d.

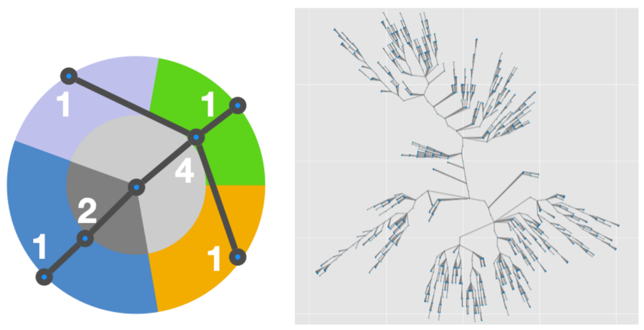


Figure 3: **Left:** Illustration of initial radial layout of RT_C. The number indicate the proportion of wedge sectors, determined by the size of induced subtrees of child nodes. **Right:** Initial layout of the last.FM network by RT_C.

4.2.6 Distribution Force

We define a global node distribution force: a repulsive force between every pair of nodes. We set the repulsive force between two nodes inversely proportional to the squared distance in the current layout; similar to an electrical charge between nodes: $|f_d(u,v)| = s(u,v)/||X_u - X_v||^2$, where $s(u,v)$ denotes the strength of the force between nodes and depends on the longest desired edge length adjacent to u and v. We set $s(u,v) = max_{w \in V,(u,w) \in E}\ \{l_{uw}\} * max_{w \in V,(v,w) \in E}\ \{l_{vw}\}$.

4.2.7 Node-Edge Force

Finally, we define a force between nodes and edges. This improves readability by reducing the number of instances where labels are placed over edges. This force is inversely proportional to the distance between the node and edge, acting orthogonally from the edge (evaluating to zero if the node does not project onto the edge segment or is too far from the edge): $|f_{node-edge}(v,e)| = c/d(v,e)$, where c is a constant across all pairs and $d(v,e)$ denotes the Euclidean distance between node v and edge e.

4.2.8 Final Iteration

The final iteration is needed to ensure any remaining overlaps are removed. It is possible that this step will have no work to do, but this check is necessary in order to enforce our hard constraint (C2).

In this step, we go over all pairs of overlapping nodes and move them until the overlap is repaired. To do this we check whether we can move one of the overlapping nodes so that the distance between two nodes increases without introducing any crossing and label overlap. Specifically, for each node v of the pair of nodes, we consider a square bounding

Algorithm 2: `Compact-Initialization`

```
Input: $G = (V, E)$ // The tree network
$root \in V$ // root of the tree
$\{\dots, n_v, \dots\}$ // Size of subtree rooted from $v \in V$
$\{\dots, d_v, \dots\}$ // Number of hops from the root to $v \in V$
Output: $X$ // Crossing-free initial layout for RT_C
Function Initial_Layout($G, root$):
    $X_{root} \leftarrow (0, 0)$;
    $W_{root} \leftarrow wedge(center = root,$
        $radius = 1, angle\_range = [0, 2\pi))$;
    for parent node $p \in BFS(G, root)$ do
        $\{c_1, c_2 \dots c_{|C|}\} \leftarrow reorder(children(p),$
            $mode =$ 'centralize heavy subtrees');
        $angle\_ranges = \{\dots A_{c_i} \dots\} \leftarrow$
          $partition(angle\_range(W_p); n_{c_1}, n_{c_2}, \dots)$;
        for $c_i \in C$ do
            $W_{c_i} \leftarrow wedge(center = root,$
                $radius = d_{c_i}, angle\_range = A_{c_i})$;
            $X_{c_i} \leftarrow midpoint(arc(W_{c_i}))$;
    return $X$;
```

box that has a small area. We denote the set of nodes in that bounding box by V'. We then sample some random points from that bounding box. For each of these sample points, we check whether we have an overlap-free and crossing-free drawing. If we find such a point, then we move v to that point and consider the next label overlap; see pseudocode Alg. `Final-Iteration`.

Note that in some cases this step is not needed at all, as all overlaps are removed during the force-directed layout improvement step. In other cases (e.g., larger input instances with denser subtrees) a handful of final iteration steps are needed to remove remaining overlaps.

4.3 Parallel Readable Tree Drawing

Here we describe the parallel version of the algorithm (PRT), which again maintains the hard constraints, but now also adds scalability in order to handle larger trees. The previous algorithm, RT, generates good quality layouts that satisfy all the hard constraints and optimize the soft constraints well. Additionally, RT does not require any specialized equipment and can be run on any computer. However, it is sequential in nature and does not work well for networks with more than about 5000 nodes. The parallel tree version takes advantage of opportunities to speed up some of the necessary computations. As before, there are two variants: Edge Length Initialized Parallel Readable Tree (PRT_L) algorithm and Compactness Initialized Parallel Readable Tree (PRT_C) algorithm that emphasize the preservation of desired edge lengths and compactness.

Note that force calculations in the PRT algorithm have an inherent dependence on neighbors and non-neighbors (i.e., collision/edge forces for a node depend on the coordinates of other nodes). If such forces are computed in different threads, they cannot be seamlessly integrated. Instead of running the entire algorithm in parallel, we use the mini-batch approach, similar to BatchLayout [36]. The mini-batch technique is commonly used for parallel Stochastic Gradient Descent (SGD) training, where one gradient update has a dependency on other gradient updates [20].

The PRT algorithm follows the same workflow shown in Fig. **??**: a crossings-free initial layout, followed by customized force-directed improvement, and a final iteration the enforces the overlaps constraint.

Initialization: We create an initial layout of the tree with no edge crossings using parallelized versions of Alg. `Edge-Length-Initialization` and Alg. `Compact-Initialization`. The most time-consuming part here is determining the center node that serves as *root*. The time complexity of this step is $O(n^2)$, where n is the number of vertices in G. Thus, we fully parallelize Alg. `PRT-Center-Node`, where normalized closeness centrality [6] is computed for each vertex. Then, we take the node with the maximum score as root. Since G is a tree, we can apply BFS to compute the distance between vertices u and v as shown in line 3 of Alg. `PRT-Center-Node`.

We place the center node at the origin of the Cartesian coordinate system and iteratively traverse the tree in a BFS fashion, placing nodes so that they do not introduce edge-crossings. The runtime of this part of the algorithm is $O(n)$. Thus, finding the center node is the slowest step of Initialization, and that has been effectively parallelized so that it has minimal effect on the overall runtime.

Parallel Force-directed Improvement: We describe the parallel force-directed improvement of layout in Alg. `PRT-Force-Directed-Improvement`. In each iteration of the algorithm, we select a batch B from the set of nodes V and compute attractive and repulsive forces in parallel similar to the BatchLayout method [36]. We apply label overlap forces, edge length forces, and node-edge forces to each node and edge of B in a similar way described in Sec. 4.2.2. To compute repulsive distribution forces with respect to the non-neighboring nodes, we select $sample$ nodes at random, to speed up the process by approximate repulsive force computation [35]. For each random node w, we compute repulsive force $f_d(X_u, X_w)$ and update the temporary coordinates T_u. Note that this force computation for nodes within the same batch is independent and thus we can run it in parallel. Before updating the coordinates of a batch, we check whether it introduces edge-crossings (line 22). Even though an increased number of batches exposes more parallelism, the quality of the layout may be negatively impacted, as observed in stochastic gradient descent (SGD) [35]. We found that a batch size of 128 or 256 gives a good balance between speed and quality [36]. Since the batch size is small compared to the size of the tree, we perform sequential updates in line 23. However, we perform the edge-crossing check in parallel.

Parallel Final Iteration: This step checks for any remaining label overlaps and repairs them in parallel. The underlying edge crossings check method is similar to the parallel force computation in Alg. `PRT-Force-Directed-Improvement`. Other than the parallel edge crossings check, the algorithm is similar to Alg. `Final-Iteration`. For each overlapping pair of nodes, we sample some random points from a square bounding box that has a small area. For each of these sample points, we check whether we have an overlap-free and crossing-free drawing. When we find such a point, we move the node there.

4.4 Algorithm Parameters

Like many force-directed algorithms, our RT and PRT methods depend on several parameters. We set some default parameter values based on prior work. For example, we select effective approximation algorithms to compute multi-level Steiner trees and the number of levels in the trees proportional to size of the underlying data based on prior work [1]. We set the batch size of Alg. `PRT-Force-Directed-Improvement` equal to 128-256 as in BatchLayout [36]. Repulsive force parameters are based on those in Force2Vec [35].

We performed a small-scale parameter search for most of the remaining parameters. To determine good values for these parameters we extracted samples with 2,000 nodes from each of our datasets and analyzed the effect of parameter modification. We do this by setting up default values from pilot experiments and explore modifying one parameter at a time, while fixing the remaining ones.

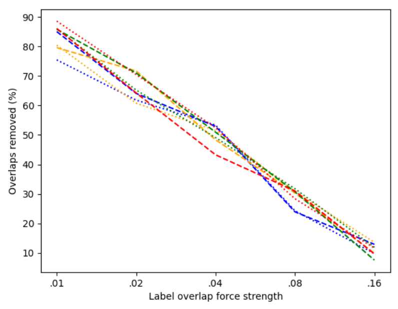
4.4.1 Different Forces

In Alg. `Force-Directed-Improvement` (lines 6-16) and Alg. `PRT-Force-Directed-Improvement` (lines 11-21), we combine several different forces acting on each node of the input network. Specifically, there are four types of forces: the label overlap force F_c to remove label overlaps, the edge length force F_l to achieve the desired edge lengths, the distribution force F_d to distribute the nodes in the drawing area uniformly, and the node-edge force F_{ne} to keep adjacent nodes closer. Each of these forces has a strength, or scaling factor, that indicates its impact on the overall force. We denote the strengths of the label overlap force, edge length force, distribution force, and node-edge force by S_c, S_l, S_d and S_{ne}, respectively. The range of the strength values is $[0-1]$. Then the total force $F(u)$ applied on a node u is calculated by the following equation:

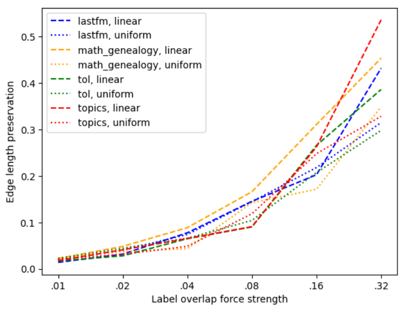
$$F(u) = S_c \cdot F_c(u) + S_l \cdot F_l(u) + S_d \cdot F_d(u) + S_{ne} \cdot F_{ne}(u)$$

Edge Length Force: The edge length force, F_l, plays a very important role in all RT and PRT variants as it corresponds to one of the two optimization goals (O1). With this in mind, we use the highest strength for this force: $S_l = 1$.

Label Overlap Force: The initialization based on edge length preservation consistently results in more overlaps that the compactness-based one. Hence to determine an appropriate strength for the label overlap force, we use Alg. `Edge-Length-Initialization` in the parameter search. Fig. 4a shows the percentage of label overlaps of all networks, with respect to the initial overlaps, after 50 iterations of Alg. `Force-Directed-Improvement`. As we increase the strength of this force from 0 to 1, overlaps decrease, while edge length preservation decreases, as illustrated in Fig. 4b. To balance this, we set the strength of label overlap force to 0.16, leaving around 10% overlaps (which are removed in the final step), while providing good edge length preservation.



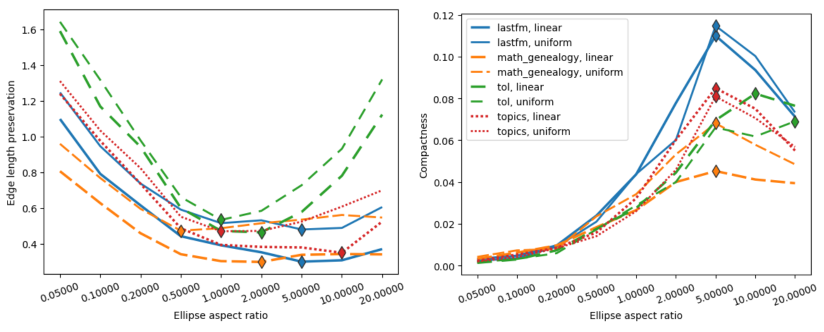
(a)



(b)

Figure 4: Illustrating the impact of the label overlap force strength. (a) shows the percentage of label overlaps with respect to the label overlap force. (b) shows the edge length preservation with respect to the label overlap force.

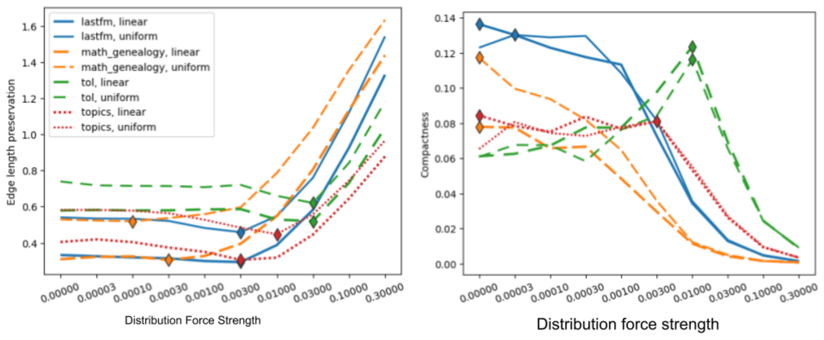
We also explore the aspect ratio of the ellipse in the label overlap force. The aspect ratio, or the parameter b defined in Sec. 4.2.4), denotes a wide elliptical collision force when $b > 1$ and a tall one when $0 < b < 1$. As shown in Fig. 5, the edge lengths can be preserved with a wide range of ellipse aspect ratios with the best compaction achieved when the aspect ratio is set to 5.



(a) (b)

Figure 5: Impact of label overlap force directions (ellipse aspect ratio) on the two quality metrics: Edge length (a) and compactness (b). Diamonds mark the best parameter setup for the specific network.

Distribution and Node-Edge Forces: The distribution force and the node-edge force play a larger role with the compactness initialization, and we experimentally determine suitable strengths with Alg. `Compact-Initialization`. As shown in Fig. 6, smaller force strength leads to better edge length preservation and better compactness for all datasets, and so we set $S_d = 0.003$



(a) (b)

Figure 6: Impact of distribution force on the two quality metrics: Edge length (a) and compactness (b). Diamonds mark the best parameter setup for the specific network.

We also explore the strength of node-edge force defined in Sec. 4.2.7. As shown in Fig. 7, both edge lengths preservation and compactness are better with a weak node-edge force. Setting node-edge force strength $S_ne = 0.1$ seems to provide a reasonable balance between the two metrics.

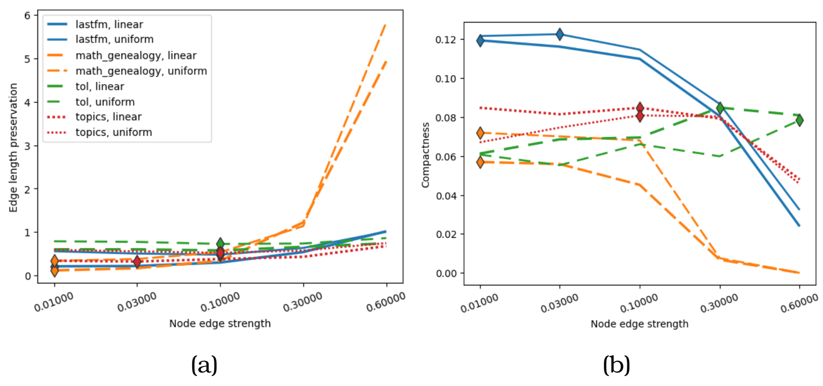


Figure 7: Impact of node-edge force on the two quality metrics: Edge length (a) and compactness (b). Diamonds mark the best parameter setup for the specific network.

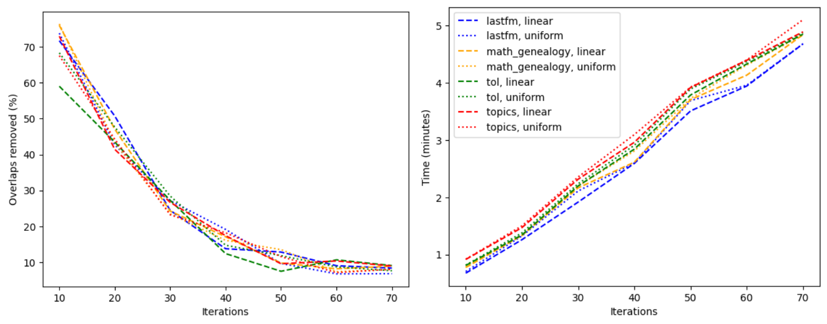
4.4.2 Number of Iterations

The force-directed improvement step of the algorithm and the final iteration perform iterative refinements; the number of iterations of each of these steps impacts the quality of the layouts and the runtime of the overall algorithm.

Number of Force-directed Iterations: The force-directed algorithm converges relatively quickly when initialized using Alg. `Compact-Initialization`. On the other hand, Alg. `Edge-Length-Initialization` needs more force-directed iterations to remove all label overlaps. Hence we analyze the impact of the number of iterations when initializing with Alg. `Edge-Length-Initialization`. As illustrated in Fig. 8a, early iterations remove many overlaps, with diminishing returns after 40-50 iterations. On the other hand, the running time increases with the number of iterations as shown in Fig. 8b. Hence, we set the number of iterations equal to 50.

Final Iteration: Since the force-directed improvement step terminates after 50 iterations, it may not have removed all overlaps. The final iteration step enforces the no-overlaps constraint (C2) by removing any remaining overlaps. Its performance depends on two parameters: the number of samples and the width of the square sample area. As illustrated in Fig. 9a, the running time increases as the number of samples increases. We can reduce the number of samples by tuning the width of the sample area. We denote the width of the sample area by the percentage of the minimal square box that contains the drawing. As illustrated in Fig. 9b, as the width of the sample area increases, the number of needed samples is reduced. Hence, we set the sample width to $0.01 - 0.02\%$ of the total area and the number of samples to 20.

We select the terminals of the multi-level Steiner tree instance ac-



(a) (b)

Figure 8: Illustrating the impact of the number of iterations on overlaps and time. (a) shows the percentage of overlaps by iterations and (b) shows the time by the number of iterations. We choose a value to balance the two considerations.

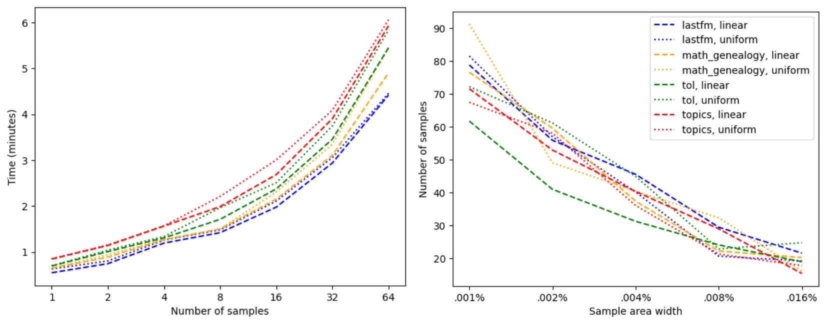
cording to the node weights: higher-level terminal sets contain heavy nodes, since the larger the weight is the more important the node is. The size of the terminal sets grows linearly. If we have n nodes and h levels then the top terminal set contains the most important n/h nodes. The next terminal set contains all the nodes of the top terminal set, together with the next n/h important nodes. We continue the process until the bottom level, when all nodes are present.

Edge Weights: Edge weights in the Last.FM network are given by the number of listeners of the corresponding pair of bands. Similarly, edge weights in the Google Topics network are the number of researchers listing the corresponding pair of topics. Edge lengths in the Tree of Life network are given by phylogenetic distance between the two endpoints. The math genealogy network is unweighted, and we use uniform edge lengths.

Desired Edge Length Settings: The edge weighs described above can be used desired edge lengths by computing the reciprocal of the original edge weights (converting similarities into dissimilarities). We have used this setting in experimenting with our algorithms but this setting is not what we report in the paper as it does not lend itself to semantic zooming and prior methods cannot handle desired edge lengths.

The simplest desired edge length setting used in the paper is the *uniform edge length setting*. We need such a setting to be able to compare the performance of our algorithms against those of prior methods (sfdp+p and CIR) that do not take edge lengths into account.

We can use the given edge weights and combine them with the multi-level Steiner tree solution to create edge lengths that work well with semantic zooming. This is the *linear edge setting* used in some of the



(a) (b)

Figure 9: Illustrating the impact of the number of samples and sample area width of the final overlap removal algorithm. (a) shows the running time by the number of samples. (b) shows the width of the sample area for the number of samples needed. We choose a value that allows for fewer samples to reduce the runtime.

experiments discussed in the paper. While in the uniform setting edge lengths are set to 200 in the linear setting the edge lengths depend on the level on which the edge first appears in the multi-level Steiner tree (which in turn depends on the underlying edge weights). The desired edge length on the lowest level is $l_{min} = 200$ and this value is increased by l_{add} each time we got to a higher level. In our examples, l_{add} varies depending on the number of levels. We use different number of levels for different networks, with more levels for larger networks. For the smallest dataset, the Last.FM network, we have only 8 levels while for the largest dataset, the math genealogy tree, there are more than 100 levels.

4.5 Interactive Interface and Prototype

In this section we describe a functional browser interface and a prototype deployment of ZMLT with two real-world datasets.

4.5.1 Layer Definition

We use a multi-level node-weighted Steiner tree (MVST) [12], defined and discussed in the Introduction to determine which level a node will appear. For our purposes we are interested in the maximum weight Steiner tree, as nodes and edges with high weights are important and should be present on the higher levels.

The number of levels in the heirarchy is a user defined parameter. We use 8 levels, with the following percentage of nodes appearing at each level: $(5, 15, 30, 40, 60, 70, 85, 100)$.

While the Steiner tree problem usually asks for the minimum cost tree, this is easy to fix by simply choosing the reciprocal of the weights (assuming all weights are positive and greater than 1). This approach can also be extended to unweighted graphs, where node weights can be assigned based on structural graph importance, e.g. node degrees or betweenness centrality.

4.5.2 Web Browser Interface

The output of the algorithms contain the positions of all the nodes in all the levels of the graph. We modify the node-link representation into a map-like one using *GMap* [17]. Finally, we extract GeoJSON files and use OpenLayers [39] to display the data in the browser. We briefly describe the process.
Node Clustering: By default, we use MapSets [26] to cluster the graph and show the clusters as contiguous regions on the map, although other GMap [17] options are also available.
Generating Map Layers: GeoJSON [10] is a standard format for representing simple geographical features, along with their non-spatial attributes. We split all geometric elements of the map into *nodelayer*, *edgelayer*, and *clusterlayer*. Each of the map layers is composed of many attributes. For example, nodelayer contains node-ID, coordinates, font-name, font-size, height, width, label and weights for all nodes.
JavaScript Application: For rendering and visualizing the map layers we use OpenLayers [39], a JavaScript library for displaying map data in the browser. This visualization requires additional work to show the map elements in a multi-layer fashion. Node attributes, edge attributes and zoom level are used to determine the appropriate levels in the multi-level visualization.
Interactions: As an interactive visualization tool, we enable panning and zooming, as well as clicking on nodes and edges to see details such as level and weight.

4.5.3 Prototype ZMLT Deployment

In this section we evaluate the DELG and CG algorithms on two real-world graphs: (i) the Google Topics Graph [9] capturing relations between research topics extracted from Google Scholar with 5947 nodes and 26695 edges; and (ii) the Last.fm Graph [17] showing relations between musical artists with 2588 nodes and 28221 edges.

The Google Topics graph is obtained from Google Scholar academic research profiles. The nodes of the graph are research topics, with weights corresponding to the number of people reporting to work on them. Edges are placed between pairs of topics that co-occur in the profiles.

In the Google Topics graph, there are many high degree nodes, for example, the maximum degree is 153. It is hard to remove label overlaps when one vertex has a large number of neighbors. Hence, we removed a few edges and made the maximum degree equal to 30. In the original graph, the number of nodes was 5943, after reducing the degree the number of nodes becomes 5483. In the degree reduction method, we compare the edge weight with the weight of the end vertices, and we compute a score. If that score is smaller than a cut-off then we delete the edge. Specifically, we use the following formula:

$$\frac{weight(e) - \frac{weight(u) \times weight(v)}{W}}{\sqrt{\frac{weight(u) \times weight(v)}{W}}} < cutoff$$

Here, $e = (u, v)$ is an edge and $W = \sum_{v \in V} weight(v)$. We keep the value of cut-off equal to 50 for our dataset.

The Last.fm graph, is based on data from the last.fm website, an Internet radio and music community website with a recommender system based on user listening habits. The nodes are popular musical artists with weights corresponding to the number of listeners. Edges are placed between pairs of similar artists, based on listening habits.

Both of these graphs can be interactively explored in our prototype ZMLT implementation: `http://uamap-dev.arl.arizona.edu:8086/`.

We compare our two ZMLT algorithms to an implementation relying on an off-the-shelf tool using both the Google Topics graph and the Last.fm graphs as defined above.

Direct Approach: The goals of ensuring a crossing-free output and labeling all nodes without overlaps can be achieved using the off-the-shelf *Circular Layout* offered by *yED*. This algorithm produces layouts that emphasize group and tree structures within a network by drawing trees in a radial tree layout fashion [41]. We produced this layout by drawing the last layers of the hierarchies, namely, T_8 using Circular Layout with BCC Compact and the "consider node labels" feature. Then we extracted the subtrees of the hierarchy. In the following we call this procedure the *Direct Approach*.

Quality Metrics: We use the following two metrics to compare our outputs and those from the direct approach:

- *Desired edge length (DEL)* evaluates the normalized desired edge lengths in each layer. Since higher levels include the most important nodes and edges they correspond to larger nodes and longer edges. Given the desired edge lengths $\{_{ij} : (i, j) \in E\}$, defined in **??**, and coordinates of the nodes X in the computed layout, we evaluate DEL with the following formula:

$$\texttt{DL} = \sqrt{\frac{1}{|E|} \sum_{(i,j)\in E} \left(\frac{||X_i - X_j|| - l_{ij}}{l_{ij}} \right)^2} \tag{1}$$

This measures the root mean square of the relative error as in [2] and produces a positive number, with zero corresponding to perfect realization.

- *Compactness* measures the ratio between the sum of the areas of labels (the minimum possible area needed to draw all labels without overlaps) and the area of the actual drawing (measured by the area of the smallest bounding rectangle).

$$\texttt{CM} = \frac{\sum_{v\in V} \text{label\_area}(v)}{(X_{max,0} - X_{min,0})(X_{max,1} - X_{min,1})} \tag{2}$$

CM scores are in the range $[0, 1]$, where 1 corresponds to perfect area utilization.

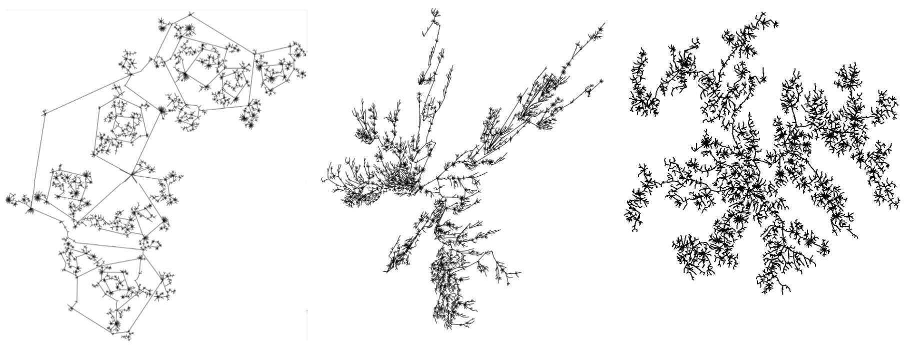
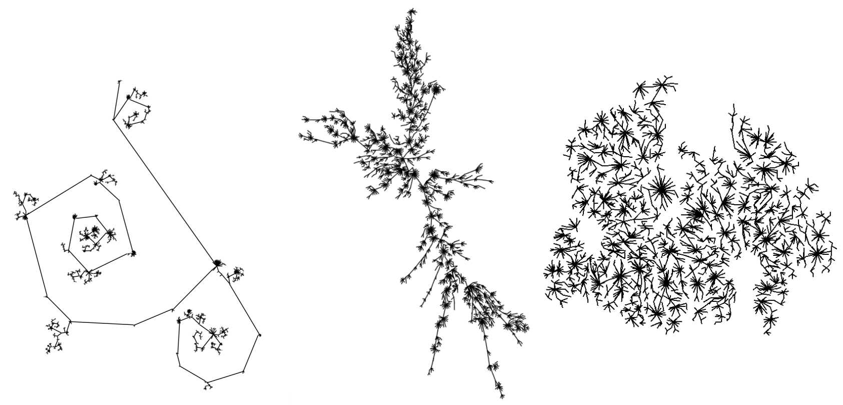

(a) Direct Approach (b) RT_L (c) RT_C

Figure 10: Comparison of the tree layout structure of the Google Topics graph drawn with the Direct Approach, our DesirableEdgeLengthguided (DELG) Algorithm and our Compactnessguided (CG) Algorithm.

Results: We show the layouts created by the three algorithms for the Google Topics graph in Figure 10 and for the Last.fm graph in Figure 11. These figures highlight some significant differences which stand out visually. For example, in order to achieve compact layouts, the direct approach places nodes in a spiral fashion, which is not a good representation of the underlying graphs.



(a) T_8 (b) T_8 (c) T_8

Figure 11: Comparison of the tree layout structure of the Last.fm graph drawn with the Direct Approach, our DesirableEdgeLengthguided (DELG) Algorithm and our Compactnessguided (CG) Algorithm.

The metric analysis in Table 2 and Table 4 shows that the two ZMLT algorithms better utilize the drawing area and better realize the desired edge lengths. The tables report the values for the two measures computed on each of the 8 trees in the graph hierarchy (representing the graphs at different levels of detail). Low values in desired edge length preservation and high values in compactness correspond to better results. According these measurements, the ZMLT algorithms are consistently better than the direct approach.

Error Analysis: From the measures in Table 2 and Table 4 as well as the overview of tree layouts, we can see that the two ZMLT algorithms favor different objectives: the RT_L algorithm performs better in desired edge length preservation while the RT_C algorithm performs better in compactness. Here we take a closer look at the results returned by these two algorithms and try to analyze their respective failure modes.

First, we focus on desired edge length preservation. Figure 12 colors individual edges in the layout by their relative error. Recall that we use the relative error to measure the desired length preservation in Equation 1. For each edge $e_{ij} \in E$, it measures the discrepancy between the actual edge length $||X_i - X_j||$ in the drawing and the given desired edge length l_{ij}:

$$\text{relative\_error} = \frac{||X_i - X_j|| - l_{ij}}{l_{ij}}$$

With the RT_L algorithm, we observed from the left layout and histogram in Figure 12 that most edges are drawn with their desired edge lengths, thanks to its edge-length guided initialization. We can see several extensively stretched edges (colored in blue in Figure 12), but since most of the other edges are drawn near perfectly in terms of edge length, the drawing has a low variation in relative error. On the other hand, with the RT_C algorithm the error is well distributed, but when comparing to RT_L the edge lengths tend to deviate more from the desired lengths. We suspect this is due to how the RT_C algorithm attempts to remove label overlaps: we can see that in dense regions the edges are more likely to be stretched, whereas on the periphery the edges tend to be compressed.

Next we look at the compactness of the two ZMLT layouts. Consider the difference between the two algorithms in their rendering of the region around the 'Artificial Intelligence' node in the maps, as shown in Figure 13. From the layout overview in Figure 13, we can already see that the RT_L algorithm uses space less efficiently, as it leaves more empty space, for example on the lower right of the whole layout. The 'Artificial Intelligence' node, circled in blue in Figure 13, is close to multiple heavy subtrees such as those from the nodes 'natural language processing', 'machine learning' and 'computer vision'. On the RT_C algorithm the heavy trees are distributed evenly, due to its good initial layout. The RT_L algorithm, on the other hand, placed all heavy branches in the top left quadrant, resulting in a less efficient use of drawing area, especially in the top right region or bottom left region around the 'Artificial Intelligence' node.

5 Proposal

5.1 Interactions

We propose adding some new interactions to the current system. These will allow users to interact with the graph more and find things that interest them.

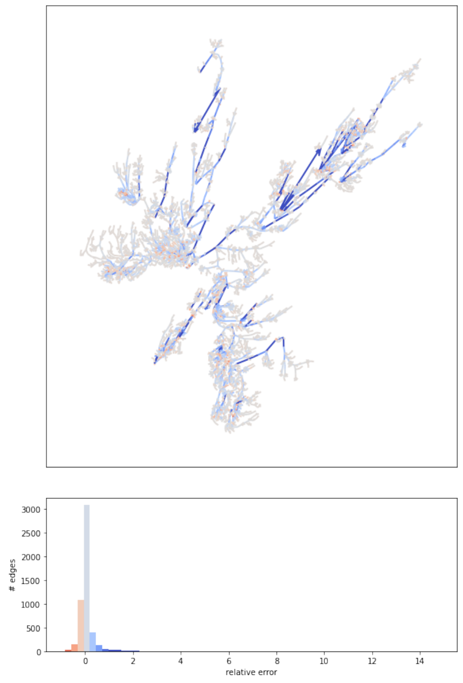
5.1.1 Path Finding

The multi-layer network gives an interesting take on path finding. We can give several different ways to find paths in a network such as this.

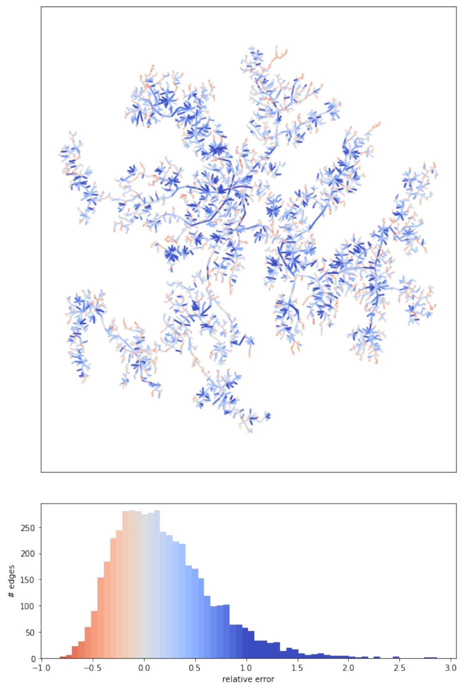
Shortest Path on the Tree: The first option is a simple shortest path along the defined Steiner tree. This is the simplest option and will give users a way to find paths along the edges shown.

Shortest Path on the Underlying Graph: The next option will give users the direct path with all edges given. The underlying graph may contain a shorter path than the tree, so this will give the real shortest path. However, it may use edges not shown in our tree structure.

Path through the top: Another option is to give users a path that goes through a high level node. This is akin to google maps who will often have several options of paths, one of which includes getting to a



(a) RT_L



(b) RT_C

Figure 12: Analysis of failure modes in desired edge length preservation. **Top left:** In the layout of Topic graph computed by the RT_L algorithm, most edges are drawn with their desired edge lengths, while some of the edges are drawn too long (colored in blue) and almost no edges are drawn too short (in red). **Bottom left:** A histogram of relative errors in desired edge length preservation, colored in the same way as the layout. **Top right and Bottom right**: The same visual analysis on the Topic Graph layout returned by the RT_C algorithm, where highway edges often stretched (blue) and edges closer to leaf nodes are often compressed (red).

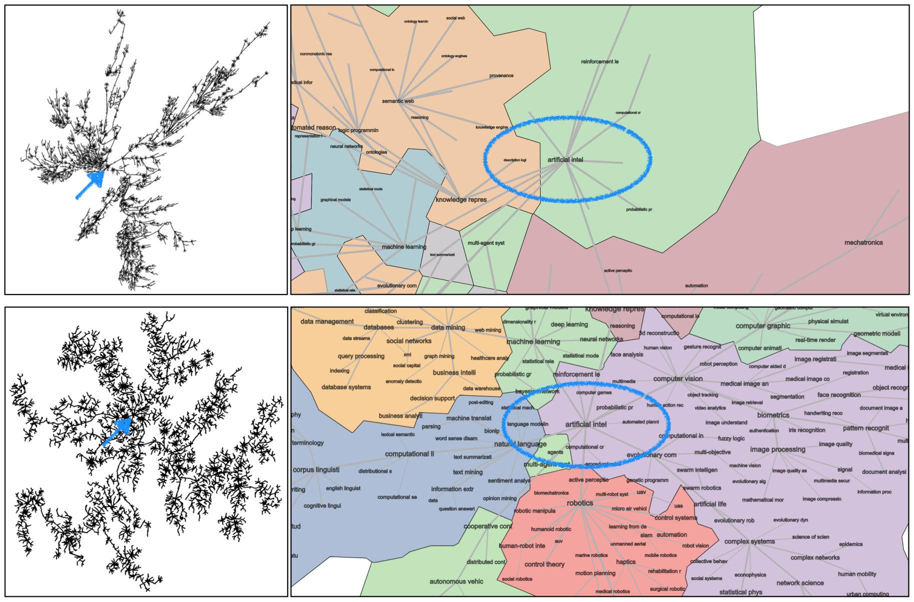


Figure 13: Analysis of failure modes in layout compactness. **Top:** The topic graph layout returned by the RT_L algorithm has more unused space and fails to distribute heavy subtrees around the 'Artificial Intelligence' node (pointed and circled in blue) evenly. **Bottom:** The RT_C algorithm generates a more balanced layout and is able to distribute subtrees more evenly.

larger street. This may also help users when the path jumps between clusters to understand where that is happening.

5.1.2 Cluster Labeling

Users are often interested in what the clusters might mean. While we can often make good guesses based on the nodes in the cluster, having a label based on some simple NLP techniques will help users navigate and understand the graph.

Some simple work has been done on this using just the node labels, however, this has not given back good results. We will work with node metadata as well to find a better way to label the clusters.

5.1.3 Node Types

Finally, we have found a use case and users that would like to be able to see different types of nodes. In this case, we have two different types of nodes to show. We will allow users to look at different types of nodes and define different shapes or colors for different types. More interactions may come from this as the user continues to work with the system.

5.2 User Evaluations

We also propose to work on more qualitative evaluations.

5.2.1 Important tasks to test

We will base these tasks from user interaction studies that give important tasks for graph understanding.

Users should be able to find nodes easily, especially after seeing a node before. We do include a search bar function, but finding similar nodes should be straightforward.

5.2.2 Comparing against other Methods

Current other systems for graph visualization include yEd and Gephi. Some of the algorithms that have been used previously are written out in the paper, including the circular layout. Comparisons with experts who use yEd or Gephi will give us insight into how our system can be used in data exploration. Comparisons with non-experts will give us insight into how our systems capitalize on previous experiences with maps and whether this helps people understand the data more thoroughly.

5.2.3 Parameter tuning

We made several design choices that could be evaluated. We create some number of levels for each graph, sometimes this was a large number and sometimes it was as small as 8. Generally, a larger number seemed to give a better zooming experience. However, having such a large number of levels can make it less clear on how important or central one node is in comparison to another. To determine what a good number of levels is for different tasks will be done using user testing, based on our important tasks laid out above.

Additionally, our force directed algorithm has a lot of parameters to work on tuning. Parameters generally involve how strong each force is. Tuning these may give us a better layout and, therefore, a better visualization.

References

[1] Reyan Ahmed, Patrizio Angelini, Faryad Sahneh, Alon Efrat, David Glickenstein, Martin Gronemann, Niklas Heinsohn, Stephen Kobourov, Richard Spence, Joseph Watkins, and Alexander Wolff. Multi-level Steiner trees. *ACM Journal of Experimental Algorithmics*, 24(1):2.5:1–2.5:22, 2019.

[2] Reyan Ahmed, Felice De Luca, Sabin Devkota, Stephen Kobourov, and Mingwei Li. Graph drawing via gradient descent, $$(gd)^2$$.InDavidAuberandPavelValtr, editors, GraphDrawingandNetworkVisualization, pages3−−17, Cham, 2020.SpringerInternationalPublishing.*

[3] Alessio Arleo, Walter Didimo, Giuseppe Liotta, and Fabrizio Montecchiani. A distributed multilevel force-directed algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):754–765, 2018.

[4] Christian Bachmaier, Ulrik Brandes, and Barbara Schlieper. Drawing phylogenetic trees. In Xiaotie Deng and Dingzhu Du, editors, *ISAAC'05: Proceedings of the International Symposium on Algorithms and Computations*, Lecture Notes in Computer Science, pages 1110–1121. Springer, 2005.

[5] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. In *International AAAI Conference on Web and Social Media*, 2009.

[6] Alex Bavelas. Communication patterns in task-oriented groups. *The journal of the acoustical society of America*, 22(6):725–730, 1950.

[7] Kevin W. Boyack, Richard Klavans, and Katy Börner. Mapping the backbone of science. *Scientometrics*, 64(3):351–374, 2005.

[8] Ulrik Brandes and Christian Pich. Eigensolver methods for progressive multidimensional scaling of large data. In *Graph Drawing*, pages 42–53. Springer, Springer, 2007.

[9] Randy Burd, Kimberly Espy, Iqbal Hossain, Stephen Kobourov, Nirav Merchant, and Helen Purchase. GRAM: Global research activity map. In *Intl. Conference on Advanced Visual Interfaces (AVI)*, pages 31:1–31:9. ACM, 2018.

[10] Howard Butler, Martin Daly, Allan Doyle, Sean Gillies, Tim Schaub, and Christopher Schmidt. The geojson format specification. *Rapport technique*, 67, 2008.

[11] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (OGDF). *Handbook of Graph Drawing and Visualization*, pages 543–569, 2011.

[12] Faryad Darabi Sahneh, Alon Efrat, Stephen Kobourov, Spencer Krieger, and Richard Spence. Approximation Algorithms for the Vertex-Weighted Grade-of-service Steiner Tree Problem. *arXiv e-prints*, page arXiv:1811.11700, Nov 2018.

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

[14] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, Gordon Woodhull, Short Description, and Lucent Technologies. Graphviz — open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer, 2001.

[15] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[16] P. Gajer, M. Goodrich, and S. Kobourov. A fast multi-dimensional algorithm for drawing large graphs. *Computational Geometry: Theory and Applications*, 29(1):3–18, 2004.

[17] E. R. Gansner, Y. Hu, and S. Kobourov. GMap: Visualizing graphs and clusters as maps. In *2010 IEEE Pacific Visualization Symposium (PacificVis)*, pages 201–208, 2010.

[18] Emden R. Gansner and Yifan Hu. Efficient node overlap removal using a proximity stress model. In Ioannis G. Tollis and Maurizio Patrignani, editors, *Graph Drawing*, pages 206–217, Berlin, Heidelberg, 2009. Springer.

[19] Emden R. Gansner and Stephen C. North. Improved force-directed layouts. In *Proceedings of the 6th International Symposium on Graph Drawing*, Graph Drawing '98, pages 364–373. Springer, 1998.

[20] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[21] R. Hadany and D. Harel. A multi-scale algorithm for drawing graphs nicely. *Discrete Applied Mathematics*, 113(1):3–21, 2001.

[22] Steffen Hadlak, Heidrun Schumann, and Hans-Jörg Schulz. A survey of multi-faceted graph visualization. In *EuroVis (STARs)*, pages 1–20, 2015.

[23] Yifan Hu. Efficient, high-quality force-directed graph drawing. *Mathematica Journal*, 10(1):37–71, 2005.

[24] Chanwut Kittivorawang, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Fast and flexible overlap detection for chart labeling with occupancy bitmap. In *IEEE VIS Short Papers*, 2020.

[25] Dmitry Kobak and Philipp Berens. The art of using t-sne for single-cell transcriptomics. *Nature communications*, 10(1):1–14, 2019.

[26] Stephen Kobourov, Sergey Pupyrev, and Paolo Simonetto. Visualizing Graphs as Maps with Contiguous Regions. In N. Elmqvist, M. Hlawitschka, and J. Kennedy, editors, *EuroVis - Short Papers*. The Eurographics Association, 2014.

[27] Yehuda Koren, Liran Carmel, and David Harel. Ace: A fast multiscale eigenvectors computation for drawing huge graphs. In *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, pages 137–144. IEEE, 2002.

[28] Yao Yang Leow, Thomas Laurent, and Xavier Bresson. Graphtsne: A visualization technique for graph-structured data. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[29] Martin Luboschik, Heidrun Schumann, and Hilko Cords. Particle-based labeling: Fast point-feature labeling without obscuring other visual features. *IEEE transactions on visualization and computer graphics*, 14(6):1237–1244, 2008.

[30] Kim Marriott, Peter Stuckey, Vincent Tam, and Weiqing He. Removing node overlapping in graph layout using constrained optimization. *Constraints*, 8(2):143–171, 2003.

[31] Kevin Mote. Fast point-feature label placement for dynamic visualizations. *Information Visualization*, 6(4):249–260, 2007.

[32] Lev Nachmanson, Arlind Nocaj, Sergey Bereg, Leishi Zhang, and Alexander Holroyd. Node overlap removal by growing a tree. pages 33–43, 2016.

[33] Quang Vinh Nguyen and Mao Lin Huang. A space-optimized tree visualization. In *IEEE Symposium on Information Visualization, 2002. INFOVIS 2002.*, pages 85–92, 2002.

[34] Carolina Nobre, Miriah Meyer, Marc Streit, and Alexander Lex. The state of the art in visualizing multivariate networks. In *Computer Graphics Forum*, volume 38, pages 807–832, 2019.

[35] Khaledur Rahman, Majedul Sujon, and Ariful Azad. Force2Vec: Parallel force-directed graph embedding. In *Intl. Conference on Data Mining (ICDM)*, pages 442–451. IEEE, 2020.

[36] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. BatchLayout: A batch-parallel force-directed graph layout algorithm in shared memory. In *2020 IEEE Pacific Visualization Symposium (PacificVis)*, pages 16–25. IEEE, 2020.

[37] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, SE-7(2):223–228, 1981.

[38] H. Schulz. Treevis.net: A tree visualization reference. *IEEE Computer Graphics and Applications*, 31(6):11–15, Nov 2011.

[39] The OpenLayers Dev Team. *OpenLayers*, 2020 (accessed December 22, 2020).

[40] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[41] Roland Wiese, Markus Eiglsperger, and Michael Kaufmann. yfiles visualization and automatic layout of graphs. In *Graph Drawing Software*, pages 173–191. Springer, 2004.

Algorithm 3: Force-Directed-Improvement

```
Input: $G = (V, E)$ // The tree network
$X$ // Crossing-free initial layout
$niter$ // Number of iterations
Output: $X$ // Improved crossing-free layout
Function Force_Directed_Improvement($G, X$):
    for $i = 1, 2, \ldots, niter$ do
        for each node $u \in V$ do
            // Label overlap force
            for each node $v \in$ collision_region(u) do
                $T_v \leftarrow \Delta T_v + f_c(X_v, X_u)$ ;
            end
            // Edge length force
            for each neighbor $v$ of $u$ do
                if $length(X_u, X_v) > l_{uv}$ then
                    $T_v \leftarrow T_v + f_a(X_u, X_v)$;
                end
                else
                    $T_v \leftarrow T_v - f_r(X_u, X_v)$;
                end
            end
            // Distribution force
            for each node $v$ of $u$ do
                $T_v \leftarrow T_v - f_d(X_u, X_v)$ ;
            end
            // Node-edge force
            for each neighbor $v$ of $u$ do
                $T_v \leftarrow T_v - f_{node-edge}(X_u, X_v)$ ;
            end
            // Maintaining no edge crossings
            if $T_v$ does not introduce edge-crossing then
                $X_v \leftarrow X_v + T_v$;
            end
        end
    end
    return $X$;
```

```
Algorithm 4: Final-Iteration
  Input: $steps$ // Sample size
  $size$ // Width of sample area
  Output: $X$ // A crossing-free layout with reduced label
     overlaps
  Function Final_Overlap_Removal($steps, size$):
      for $u, v \in V \times V$ s.t. $u$ and $v$ overlaps do
          for $k = 1, 2, \cdots, steps$ do
              $r = random(0, 1)$;
              $\Delta X_u = (r * size) - (size/2)$;
              if $\Delta X_u$ does not introduce crossing and new overlap
               then
                  $X_u \leftarrow X_u + \Delta X_u$
              end
              if $u$ and $v$ overlaps then
                  $\Delta X_v = (r * size) - (size/2)$;
                  if $\Delta X_u$ does not introduce crossing and new overlap
                   then
                      $X_v \leftarrow X_v + \Delta X_v$
                  end
              end
          end
      end
      return $X$;
```

```
Algorithm 5: PRT-Center-Node
  Input: $G = (V, E)$ // The tree network
  Output: $n_c$ // center node id $n_c$
  Function PRT_Center_Node(G):
      $C_u \leftarrow 0.0, \forall u \in V$
      for each node $u \in V$ in parallel do
          $\mathcal{D} \leftarrow \sum_{v \in V} dist(u, v)$
          $C_u \leftarrow \frac{|V|}{\mathcal{D}}$
      end
      $n_c \leftarrow \arg\max_u C_u$
      return $n_c$
```

Algorithm 6: `PRT-Force-Directed-Improvement`

```
Input: $G = (V, E)$ // The tree network
$X$ // Crossing-free initial layout
$batch$ // No.  of vertex batches form V
$samples$ // Sample size
$niter$ // Number of iterations
Output: $X$ // Improved crossing-free layout
Function PRT_Force_Directed_Improvement($G, X$):
    for $i = 1, 2, \ldots, niter$ do
        $T \leftarrow \{0\}^{|V| \times 2}$
        Partition $V$ into $B = \lceil \frac{|V|}{batch} \rceil$ batches
        for each batch $B \in V$ do
            for each node $u \in B$ in parallel do
                // Label overlap force
                for each node $v \in$ collision_region(u) do
                    $T_v \leftarrow \Delta T_v + f_c(X_v, X_u)$
                end
                // Edge length force
                for each neighbor $v$ of $u$ do
                    if $length(X_u, X_v) > l_{uv}$ then
                        $T_u \leftarrow T_u + f_a(X_u, X_v)$
                    end
                    else
                        $T_u \leftarrow T_u - f_r(X_u, X_v)$
                    end
                end
                // Distribution force
                for a random node $w$ upto samples times do
                    $T_u \leftarrow T_u - f_d(X_u, X_w)$
                end
                // Node-edge force
                for each neighbor $v$ of $u$ do
                    $T_u \leftarrow T_u - f_{node-edge}(X_u, X_v)$
                end
                // Maintaining no edge crossings
                if $T_u$ does not introduce edge-crossing then
                    $X_u \leftarrow X_u + T_u$
                end
            end
        end
    end
    return $X$;
```

Tree	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Crossings	14	55	173	264	414	514	709	886

Table 1: Number of crossings in the layouts of Topics graph obtained using prism.

	Desired Edge Length			
	DIR	RT_L	RT_C	PR
T_1	0.56	**0.03**	0.38	1.84
T_2	1.26	**0.12**	0.40	1.28
T_3	1.64	**0.13**	0.39	1.01
T_4	1.84	**0.21**	0.41	0.92
T_5	2.55	**0.28**	0.42	0.81
T_6	1.98	**0.41**	0.43	0.78
T_7	2.36	**0.44**	0.45	0.74
T_8	1.93	**0.40**	0.51	0.72

	Compactness		
DIR	RT_L	RT_C	PR
1e-5	**0.019**	0.010	0.005
1e-5	0.008	**0.009**	0.008
1e-5	0.011	**0.012**	0.013
9e-6	0.012	**0.021**	0.017
7e-6	**0.014**	0.011	0.026
1e-5	0.014	**0.015**	0.031
6e-6	0.015	**0.020**	0.038
7e-6	0.011	**0.026**	0.034

Table 2: Metric-based Topics graph layout evaluation of the Direct Approach (DIR), DesirableEdgeLengthguided (DELG) Algorithm and Compactnessguided (CG) Algorithm. **Left:** Desired Edge Length, evaluated through Equation 1. Values ranges from 0 (best) to positive infinity (worst). **Right:** Compactness evaluated through Equation 2 values range from 0 (worst) to 1 (best).

Tree	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Crossings	3	12	32	37	68	81	111	147

Table 3: Number of crossings in the layouts of Last.fm graph obtained using prism.

	Desired Edge Length			
	DIR	RT_L	RT_C	PR
T_1	1.262	**0.04**	0.43	0.77
T_2	1.484	**0.05**	0.50	0.72
T_3	1.603	**0.06**	0.51	0.65
T_4	1.727	**0.09**	0.51	0.62
T_5	1.847	**0.10**	0.53	0.60
T_6	1.907	**0.12**	0.54	0.58
T_7	1.976	**0.12**	0.56	0.57
T_8	2.061	**0.13**	0.60	0.56

	Compactness		
DIR	RT_L	RT_C	PR
1.5e-6	0.017	**0.018**	0.005
1e-6	0.015	**0.025**	0.01
1e-6	0.015	**0.022**	0.017
6e-7	0.014	**0.026**	0.019
5e-7	0.012	**0.032**	0.020
4e-7	0.011	**0.034**	0.022
4e-7	0.010	**0.038**	0.025
3e-7	0.010	**0.041**	0.045

Table 4: Last.fm graph layout evaluation of the Direct Approach (DIR), DesirableEdgeLengthguided (DELG) Algorithm and Compactness-guided (CG) Algorithm. **Left:** Desired Edge Length, evaluated through Equation 1 where the value ranges from 0 (best) to positive infinity (worst). **Right:** Compactness evaluated through Equation 2 where the value ranges from 0 (worst) to 1 (best).