Scalable Methods for Readable Tree Layouts

Abstract

Large tree structures are ubiquitous, however, scalable, easy-to-read tree layouts are difficult to achieve. Real-world relational datasets usually have information associated with nodes (e.g., labels or other attributes) and edges (e.g., weights or distances) that need to be communicated to the viewers. We describe three new algorithms for constructing readable tree layouts. We evaluate the performance of the new methods by comparison with related earlier approaches using several real-world datasets.

1. Introduction

Many real-world datasets can be represented by a network where each node represents an object and each link represents a relationship between objects; e.g., the tree of life captures the evolutionary connections between species and a research topics network captures relationships between research areas. Abstract networks can be modeled by node-link diagrams, with points representing nodes and segments/curves representing the edges. However, real-world datasets have labels associated with nodes and attributes such as edge lengths that are not captured in node-link diagrams. For example, the tree of life has species names as node labels and evolutionary distance between the corresponding species as edge data. These graphs would benefit from a visualization that shows these labels and captures the desired edge lengths. However, just adding labels to an existing layout will result in an unreadable visualization with many overlapping labels. One could scale the layout to remove such overlaps, but this could blow up the drawing area to unmanageable size. Another reasonable strategy is to apply a specialized overlap removal algorithm. This avoids the blowup in area, but results in layout changes: not just changing edge lengths, but also changing the topology (e.g., breaking up clusters, or introducing new edge crossings).

We propose an algorithmic framework for creating readable tree layouts and evaluate it on several real-world datasets. Since we are laying out trees, a crossing-free layout is possible and desirable. This gives us two hard *constraints*: (C1) No edge crossings, (C2) No label overlaps. We also consider three additional desirable properties that the algorithms *optimize*: (O1) desired edge lengths, (O2) drawing area, and (O3) efficient runtime. Realizing pre-specified, desired edge lengths is important to many real-world datasets, but is not taken into account by most graph and tree layout algorithms. Keeping track of the area required (e.g., by comparing to the sum of areas of all labels), makes it clear that simply scaling up a given layout until labels do not overlap results in unusable layouts with areas that are 4-6 orders of magnitude greater than needed. Finally, the scalability of the algorithm is important when dealing with larger datasets containing hundreds of thousands of nodes.

Figure 1: A map generated from a readable tree layout of a real-world research topics network.

Despite there being more than 300 algorithms for drawing trees [1], none can guarantee the two constraints (no crossings, no overlaps), while also optimizing desired edge lengths, area, and runtime. With this in mind, we propose three algorithms: one guided by realizing desired edge lengths, one guided by minimizing the required drawing area, and one that optimizes runtime. The layouts obtained from these algorithms guarantee the two constraints, no edge crossings or label overlaps. We evaluate these algorithms with 4 different real-world datasets: from trees with 2,588 nodes, up to trees with 100,347 nodes. We also compare our new algorithms against state of the art general graph layout and tree layout algorithms, by relaxing some of the constraints.

We also show the utility of the proposed algorithmic

framework for drawing readable tree layouts to visualizing general graphs. There are many algorithms for extracting *important* trees from a given graph: from the minimum spanning tree and maximum spanning tree, to Steiner trees and network backbone trees. In particular, motivated by people's familiarity with maps [2], a multi-level Steiner tree algorithm can be used to create a level-of-detail representation of the underlying graph, which can be used to drive a multi-level, interactive, map-like representation; see Fig. 1.

2. Related Work

Tree and Graph Layout Algorithms: Drawing trees has a rich history: Treevis.net [1] 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 [3]. 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.

Most general graph layout algorithms use a force-directed [4, 5] or stress model [6, 7] and provide a single static drawing. The force-directed model works well for small graphs, but does not scale to large networks. Speedup techniques employ a multi-scale variant [8, 9] or use parallel and distributed computing architecture such as VxOrd [10], BatchLayout [11], and MULTI-GILA [12].

Libraries such as GraphViz [13] and OGDF [14] provide graph layout algorithms but they do not support interaction, navigation, and data manipulation. Visualization toolkits such as Gephi [15] and yEd [16] 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 graphs.

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 [17, 18]. 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.* [19] 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 [20], Gansner and Hu [21], and Nachmanson *et al.* [22] describe overlap-removal techniques with better aspect ratio and modest additional area. However, none of these approaches (except the straightforward scaling), can guarantee that no crossings are added when starting with a crossings-free input. Placing labeled nodes without overlaps has also been studied [23–25], but these approaches do not guarantee crossing-free layouts of the underlying graphs.

3. Readable Tree Layout Framework

We begin with a review of the high-level objectives for readable tree layouts, the need for new algorithms, and the proposed algorithmic framework.

High-level objectives: The main goal of this paper is to describe our algorithmic framework for visualizing large labeled trees while realizing given edge lengths. In particular, we aim to generate "readable" layouts where both the tree structure and node labels are easily interpretable by a human observer. Since readability is a subjective property, we identify two strict requirements and three desirable properties of readable layouts.

- **Strict requirements (hard constraints):**

C1. **Layout readability:** Edges *must* not cross in the final layout

C2. **Label readability:** Node labels *must* not overlap and should be large enough for reading

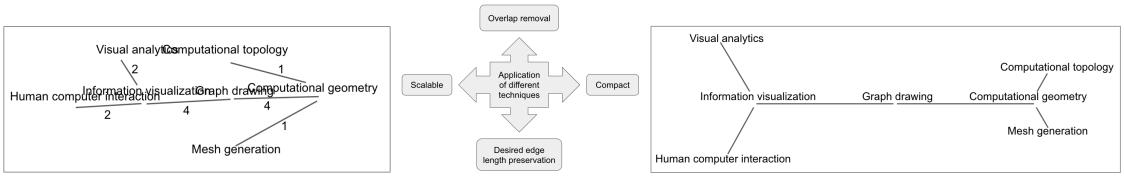
- **Optimizable desirable properties:**

O1. **Desired edge length:** The length of edges in the computed layout *should* should realize the given edge lengths

O2. **Compact drawing:** The layout *should* minimize unused space in a given drawing area

O3. **Scalability:** The algorithm *should* work for large instances and utilize modern processor architecture

While constraints C1-C2 and optimization goals O2-O3 are natural and self-explanatory, O1 requires a bit more justification. The desired edge length property is a standard requirement in instances where different edge lengths capture important information (such as evolutionary time in the tree of life). Uniform edge lengths (a special case where all desired edge lengths are the same) are preferable in cases where all edges represent the same notion of connectivity. Finally, precomputed edge lengths are useful in a multi-level representation of large trees and graphs, where higher levels contain relatively important nodes and links. As we zoom into lower levels, less important nodes and edges appear in the visualization. Such multilevel approached allow us to see the global structure but also local details. A natural way to capture the multi-level representation is to assign different desired edge lengths for different levels: the higher level edges have longer desired edge



(a) Initial layout: No edge crossing, labels overlap, desired edge length not preserved.

(b) Iterative improvements via force-directed updates.

(c) Final layout: No edge crossing, no label overlap, compact, and preserve desired edge lengths.

Figure 2: An algorithmic framework used to generate readable layouts. (a) The first step of the algorithm generates a crossing-free initial layout. (b) We then iteratively improve the layout to remove overlaps. At the same time, different algorithms aim to generate compact layouts, preserve desired edge lengths, and/or perform computations in parallel for scalability. (c) The final crossing-free and overlap-free layouts approximately optimize other desirable properties.

lengths and the lower level edges have shorter desired edge lengths. We increase the edge lengths linearly as we go from bottom level to top level, the details are provided in Sec. 7. In this paper we focus on this multi-level notion of desired edge lengths, as all our graphs and trees are large enough to require the interactive panning and zooming provided by this approach.

The need for new algorithms: While there exist many algorithms for generating tree and graph layouts, to the best of our knowledge, no existing algorithm considers all five properties mentioned above. For example, one of the most frequently used graph visualization system, GraphViz [13], has an efficient layout algorithm based on the scalable force directed placement (sfdp) algorithm [26] and then label the nodes and remove overlaps via the PRoxImity Stress Model (PRISM) [21]. The output, however, does not optimize given edge lengths and cannot ensure the crossing constraint; the examples from our 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 overlap without blowing up the drawing area. Another excellent visualization toolkit, yEd [16], provides a method that can draw trees without edge crossings and optimize compactness. However, none of the methods available in yED can realize the desired edge lengths and one cannot remove label overlap without blowing up the drawing area. This paper fills this gap in the literature by developing and making available scalable methods for readable tree layouts.

An algorithmic framework: We design a general algorithmic framework, from which we develop three new algorithms. Fig. 2 shows the generic workflow of an algorithm that takes as input a node-labeled tree with pre-specified edge lengths. For the sake of consistency, we use the same methodology to assign desirable edge lengths to all datasets here. Specifically, we extract multi-level Steiner trees and assign desired lengths proportional to the level where edge first appears.

All the algorithms in our framework begin by computing a crossing-free initial layout and maintain this property in every subsequent step. In the iterative refinement step, our algorithms employ force-directed layout improvement, tailored to removing label overlaps, realizing desired edge lengths, and minimizing area. However, simultaneously improving all three desirable properties (edge lengths, drawing area and scalability) is difficult as the individual properties could require contradictory movements. With this in mind, we develop three algorithmic variants, each of which prioritizes one of the desirable properties. The first algorithm starts with a layout with good desired edge lengths and then optimizes compactness. The second algorithm starts with a layout that has good compactness and optimizes desired edge lengths. The third algorithm speeds up the computations by processing a batch of nodes in parallel while improving compactness.

We quantitatively evaluate our three methods against each other by measuring runtime, compactness, and desired edge length realization. We also compare them against two state of the art methods, even though one of them produces crossings and we must use uniform edge lengths. Since even the smallest tree we work with has more than 2500 labeled nodes, we also make the results accessible via interactive, map-like visualizations. Specifically, using the multi-level Steiner tree hierarchy extracted from the input graph/tree, we provide semantic zooming functionality that allows us to see the global structure (high level) and local details (low level).

4. DELG Algorithm

The desired edge length guided (DELG) algorithm first computes an initial layout without crossings that preserves desired edge lengths. This layout is then iteratively refined to remove overlaps while ensuring

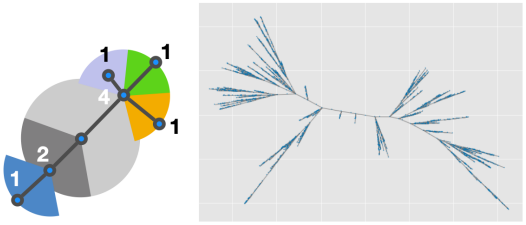


Figure 3: **Left:** Illustration of initial radial layout of DELG. The numbers indicate the proportion of wedge sectors determined by the size of induced subtrees of child nodes. **Right:** Initial layout of the last.FM graph.

good area utilization. In the last step any remaining label overlaps are removed by possibly distorting associated edge lengths, but not introducing edge crossings.

DELG Crossings-free Initialization: We start with a crossings-free initialization, similar to [27]. We select a root node and assign a wedge region (sector) to every child. Each child is then placed along angle bisector of the assigned wedge, away from its parent by the desired edge length. We continue this process recursively by letting the children assign regions to their children. Since the angular regions are unbounded, the algorithm can realize all the desired edge lengths. When we assign wedge regions to child nodes, the angles are proportional to the size of the subtree rooted from the child; see Fig. 3.

DEL and Overlap Improvement: This initial layout is crossing free and preserves desired edge lengths. However, it may contain label overlaps. To remove these overlaps, we start with this initial layout as an input and apply a customized force-directed algorithm. We consider two forces: a collision force applied to every node to remove label overlaps, and a DEL force applied to every edge to maintain the desired edge length. We further customize the force-directed algorithm to ensure that no edge crossings are introduced at any step. We test whether a move will introduce a crossing, and if it does, we do not perform this move; see Alg. 1.

The goal of the collision force is to remove label overlaps, while the edge force works to maintain the desired edge lengths. To define the collision force, each node is assigned a collision circle centered around the node. When a node u enters the collision circle of another node v then a repulsive force f_r acts between u and v: $f_r = K/d \cdot I_{d \le r}(d)$, where $I_{d \le r}$ is an indicator function for the condition $d \le r$, i.e $I_{d \le r}(d) = 1$ if $d \le r$ and 0 otherwise. Here K is a constant, by default equal to the area of the initial drawing. The force f_r is activated when the distance d from u to v is less than or equal to the collision circle radius r. The value of f_r is reciprocal to d. We apply this collision force to every node.

Algorithm 1: DEL/Overlap-improvement($steps$)

Result: A layout with improved space utilization

$X \leftarrow$ Initialization by DELG algorithm;
for $i = 1, 2, \cdots, steps$ **do**
  for *each node* v **do**
    for *each node* $u \in$ *collision_region(v)* **do**
      $\Delta X_u \leftarrow \Delta X_u + f_r(X_u, X_v)$;
  for *each edge* $e = (u, v)$ **do**
    $\Delta X_{u,v} \leftarrow \Delta X_{u,v} + f_a^e(X_u, X_v)$;
    $\Delta X_{u,v} \leftarrow \Delta X_{u,v} - f_r^e(X_u, X_v)$;
  for *each node* v **do**
    if ΔX_v *does not introduce crossing* **then**
      $X_v \leftarrow X_v + \Delta X_v$

In the initial embedding, the edge lengths are equal to the desired lengths. If we only apply the collision forces we might remove all overlaps at the expense of drastically modifying the edge lengths. To avoid this, we combine edge forces with collision forces. 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.

Overlap Removal: It is possible the previous step will leave some overlaps to remove. For every overlapping pair, we consider progressively larger local node moves that resolve the overlap. By default, we try 100 moves, sampled from a bounding box $2-3\%$ of the layout area, centered at the current position of the node; see Alg. 2.

Algorithm 2: Overlap-Removal($G(V,E), steps, size$)

Result: A label overlap-free layout

for $i = 1, 2, \cdots, |V|$ **do**
  for $j = i+1, i+2, \cdots, |V|$ **do**
    if *label i overlaps label j* **then**
      for $k = 1, 2, \cdots, steps$ **do**
        for v *in* $\{i, j\}$ **do**
          $r = random(0, 1)$;
          $del = (r * size) - (size/2)$;
          Translate node v by del;
          if *current layout is overlap-free and crossing-free* **then**
            Break the loop;

Algorithm 3: Layout Crossing Check

Result: A crossing-free layout

shuffle(V);
$niter = 12$, $k = 0.8$;
for *each node* $v \in V$ **do**
  $X_v^0 = X_v$, $i = 0$;
  while *hasCrossing(V,E) and* $i < niter$ **do**
    $X_v = X_v^0 + k^i \cdot VX_v$, i+=1;
  if i == *niter* **then**
    $X_v = X_v^0$;

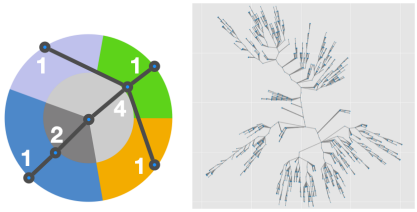


Figure 4: **Left:** Illustration of initial radial layout of CG. The number indicate the proportion of wedge sectors, determined by the size of induced subtrees of child nodes. **Right:** Initial layout of the last.FM graph by CG.

5. Compactness Guided Algorithm

We describe the second algorithm that guarantees no edge crossings and no label overlaps, but prioritizes a compact output. This algorithm begins with a modification of crossing-free layout in DELG. Starting with this layout, it employs a customized force directed algorithm that optimizes desired edge lengths and removes label overlaps, while ensuring no edge crossings are introduced.

Crossing-free Initialization: In the CG algorithm, we initialize a crossing-free layout similar to DELG, but instead of ensuring the desired edge length, we focus on more efficient use of the drawing area. Different from DELG where we assigned wedge regions to children of a node, in CG we assign the entire fan area to the children and place children along the arc of the fan. This results in a wider spread of nodes, but does not realize the desired edge lengths; see Fig. 4 and compare it with Fig. 3.

Iterative Improvement: We improve the crossings-free layout with a customized force directed algorithm that focuses on compactness, while attempting to realize desired edge lengths and avoiding label overlaps.

Desired Edge Length: To get the edge lengths closer to their desired lengths, we use an edge force that stretches compressed edges and compresses stretched edges, as described in Sec. 4.

Label Overlap Removal: To remove label overlaps, We use a collision force that is similar to the method described in Sec. 4, but with elliptical rather than circular region for collision detection. 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. In our experiments, we set it to be half of the label width. 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 will calculate and update the movement speed VX_v according to the bounding area of nodes defined by their respective radii, under the stretched layout.

Other Aesthetics: Note that edge forces and elliptical collision forces only affect the layout locally, as neither optimize global structure. To capture the global structure of the input tree, we add forces to minimize the global stress of the layout and optimize the global distribution of nodes and distances from nodes to edges.

First, we add a force to minimize stress: the difference between graph theoretical distances between pairs of nodes and the realized Euclidean distances. Formally, for every pair of nodes $u, v \in V$, the total stress is defined as $\Sigma_{u,v \in V}\, w_{uv}(||X_u - X_v||_2 - l_{uv})^2$, where $||X_u - X_v||_2$ denotes the Euclidean distance between nodes u and v, l_{uv} denotes the desired distance, and w_{uv} is a weighting coefficient. We set l_{uv} to be a power of the shortest-path distance $d_G(u,v)$ between two nodes u and v in the graph. The exponent depends on the number of hops $h(u,v)$ between two nodes. We set $l_{uv} = d_G(u,v)^{L_1 + 1/h(u,v)}$ Where L_1 is a positive constant that defines an asymptotic bound to the target distance. In our examples, we set $L_1 = 0.95$. Generally, this design of desired distance between every pair of nodes encourages a straight path between nearby nodes and a curved path between far-reaching nodes. We set the weight to be $w_{uv} = 1/d_G(u,v)^{L_2}$, where by default $L_2 = 1.3$. We minimize the stress by stochastic gradient descent - moving nodes towards the negative gradient of the stress defined above. Given that other forces go through all the n nodes once per iteration and we want to keep the number of updates in stress minimization comparable to other forces, we randomly choose n pairs of nodes to update in each iteration.

Next, we add 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: $|f_{charge}(u,v)| = s(u,v)/||X_u - X_v||^2$, where $s(u,v)$ denotes the strength of the charge between nodes and depends on the longest desired edge length adjacent to u and v. We set $s(u,v) = max_{w \in V,(u,w) \in E} \{l_{uw}\} * max_{w \in V,(v,w) \in E} \{l_{vw}\}$.

Finally, we add 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.

Crossing Check: At every iteration we check whether the layout update will introduce any new crossings. If so, for each node which would introduce new crossings, we try reducing the step size to a fraction k (e.g., $k = 0.8$) of the original. If the update at this new step size does not introduce a crossing, we make the update, but if it would still introduce a crossing, we repeat and reduce the step size further. This continues until all crossings are avoided or a maximum number of iterations (e.g. $niter = 12$) is reached. To balance the updates for each node, we shuffle the ordering of nodes in every iteration; see Alg. 3.

6. Parallel Readable Tree Drawing

The parallel BatchTree (BT) algorithm is based on the serial DELG algorithm. In the first step, we generate a crossing free initial drawing. In the second step, we use a customized force-directed method to improve the layout. Finally, we perform a post-processing step to remove any remaining label overlaps in the layout.

Initialization: We generate an initial layout of the tree with no edge crossings, similar to the initialization in Sec. 4, using the node with highest closeness centrality as root. Then, we recursively traverse the tree in breadth-first manner, placing nodes so that they do not introduce edge-crossing.

Parallel Force-directed Improvement: We describe the parallel force-directed method in Algorithm 4. 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 [11]. For each neighbor v of u, we either compute an attractive force or a repulsive force based on the current length of the edge. If the current length of neighboring nodes u and v is higher than the desired edge length l_{uv} then we apply an attractive force $f_a(X_u, X_v)$ to reduce the length (lines 8-9 of Algorithm 4), where $f_a(X_u, X_v) = \| X_u - X_v \| \cdot (X_v - X_u)$. However, if the current length is less than the desired edge length, then we apply a repulsive force $f_r(X_u, X_v)$ to push them away so that the edge length increases, where $f_r(X_u, X_v) = \frac{(X_v - X_u)}{\|X_u - X_v\|^2}$. To compute repulsive forces with respect to the non-neighboring nodes, we select s nodes at random, to speed up the process by approximate repulsive force computation [28]. For each random node w, we compute repulsive force $f_r(X_u, X_w)$ 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 15). Even though an increased batch size exposes more parallelism, the quality of the layout may be negatively impacted, as observed in stochastic gradient descent (SGD). We found that a batch size of 128 or 256 gives a good balance between speed and quality. Since the batch size is small compared to the size of the tree, we perform sequential updates in line 14. However, we perform the edge-crossing check in parallel. If it does not introduce edge-crossing, we update the coordinates with learning rate lr.

Algorithm 4: BTForce$(G(V,E), b, s, lr)$

```
   Result: improved layout X
1  X ← Initialization by DELG algorithm
2  for i = 1 to epoch do
3      T ← {0}^{|V|×2}
4      Partition V into B = ⌈|V|/b⌉ batches with b
         nodes
5      for each batch B ∈ V do
6          for each node u ∈ B in parallel do
7              for each neighbor v of u do
8                  if length(X_u, X_v) > l_uv then
9                      T_u ← T_u + f_a(X_u, X_v)
10                 else
11                     T_u ← T_u − f_r(X_u, X_v)
12             for a random node w upto s times do
13                 T_u ← T_u − f_r(X_u, X_w)
14         for each node u ∈ B do
15             if T_u does not introduce
                 edge-crossing then
16                 X_u ← X_u + lr × T_u
17     lr ← lr × 0.999
```

Parallel Label Overlap Removal: This step checks

for any remaining label overlaps and repairs them in parallel. The underlying method is similar to the parallel force computation in Algorithm 4 and in the DELG algorithm post-processing step. First, we use a batch update technique as described in Algorithm 4 to check for overlaps between pairs of labels. Then, we apply a collision/repulsive force to push them away from one another to remove the overlap. If this repulsive force does not introduce edge-crossing, we update the coordinates of the current node with the repulsive forces. Second, if needed, we deploy the post-processing step described in the overlap removal step from the DELG algorithm to remove all remaining overlaps.

7. Multi-Level Interactive Visualization

The major contribution of this paper is the algorithmic framework for readable tree layout and the three algorithms described above. However, to work with, and even to just *see* trees with thousands to hundreds of thousands of nodes, requires more than just a layout.

With this in mind, we process all trees and their layouts and provide an interactive visualization environment. The idea is to create a hierarchy of trees, starting with the input and extracting progressively smaller trees that represent more and more abstract views. We do this by computing a multi-level Steiner tree. Formally, given a node-weighted and edge-weighted graph $G = (V,E)$, we want to visualize G as 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 $E_1 \subset E_2 \subset \cdots \subset E_n \subset E$. To make the trees representative of the underlying graph we rely on a multi-level variant of the Steiner tree problem [29], where we create the 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 creates the set of progressively larger trees $T_1 \subset T_2 \subset \cdots \subset T_n \subseteq G$ using the most important (highest weight) edges.

We set the desired edge length of the lowest level equal to l_{min} and increase the desired edge lengths by l_{add} as we go from bottom to top levels. In our examples we use $l_{min} = 200$ with l_{add} varying depending on the number of levels We use different number of levels for different graphs, with higher numbers for larger graphs. For the smallest dataset, the Last.FM graph, we have only 8 levels while for the largest dataset, the math genealogy tree, there are more than 100 levels.

The Last.FM graph and the topics graph have node weights (number of listeners and number of researchers, respectively) and we use the heavy nodes in the higher levels and lighter nodes in the lower levels. For the tree of life and math genealogy tree, we set the node weight equal to the node degree.

We can consider the multi-level Steiner tree as a single-level tree with edges having different desired edge lengths. We compute the tree layouts using the algorithms described in the previous section. From the tree layout, we generate a map and clustering using the method provided in [30].

We build our prototype with OpenLayers[1]. Zooming and panning is provided via buttons, mouse scrolling, or through the mini-map.When viewing level i, all nodes at this or higher levels are labeled and there are no label-overlaps. Edge widths are determined based on their levels: higher level edges are thicker, lower level edges are thinner. User can search terms via the A search bar allows for direct queries with auto-complete suggestions and clicking on a search result recenters the map on the selected node. By default we show labels of at most 16 characters (truncating longer ones) but the full label is shown on a mouse-over event. Node attributes and edge attributes are provided when clicking on the node/edge; see Fig. 1.

8. Evaluation

Here we discuss an evaluation of the three new methods DELG, CG and BT, using 4 real-world datasets.

Prior Methods: As there are no prior algorithms that guarantee the two constraints (no crossings, no overlaps), while optimizing desired edge lengths and compactness, it can be somewhat unfair to compare against prior approaches. Nevertheless, with some careful modifications (and clarifications) we can use existing tree/graph layout algorithms in a comparison. We chose two such algorithms as described below.

GraphViz [13] can efficiently lay out a given tree or graph with sfdp [26], label the nodes and then remove overlaps via the PRoxImity Stress Model (PRISM) [21]. Note that the output does not optimize given edge lengths and does not guarantee that trees are drawn in a crossings-free manner. We denote this by sfdp+p.

The yED [16] system, provides several methods that can draw trees without edge crossings and optimize compactness. Note that yED does not optimize edge lengths and the only way to remove label overlap is to scale the drawing area. We use the *Circular Layout* (CIR) in *yED* as it produces the most compact layouts.

To provide a fair comparison we consider two settings for the evaluation: one in which we have different desired edge lengths and the other one considers uniform edge lengths (to make it possible to compare with sfdp+p

[1] `https://openlayers.org/`

(a) CIR (b) sfdp+p (c) DELG (d) CG (e) BT

Figure 5: Comparison of tree layout of the uniform Last.fm graph drawn with CIR, sfdp+p, DELG, CG, and BT.

and CIR). Note also that sfdp+p does not guarantee crossings-free drawings for trees.

Datasets: We extract 7 graphs from 4 datasets.

Last.FM Graph [30], extracted from the last.fm Internet radio station with 2588 nodes and 28221 edges. The nodes are popular musical artists with weights corresponding to the number of listeners. Edges are placed between similar artists, based on listening habits.

Google Topics Graph [31] has as nodes research topics from Google Scholar's profiles, with weights corresponding to the number of people working on them. Edges are placed between pairs of topics that co-occur in profiles. We work with two versions of this graph: one with $34,741$ nodes and $646,565$ edges, and the other a smaller subset with 5001 nodes.

Tree of Life[2], extracted from the tree of life web project. This dataset contains a node for every species, with an edge between two nodes representing the phylogenetic connection between the two. We work with two versions of this graph: one with $35,960$ nodes, and the other with $2,934$ nodes.

Math Genealogy Graph[3], every node represents a mathematician with edges capturing (advisor, advisee) relationship. We work with two versions of this graph: one with $257,501$ nodes, and the other with $3,016$ nodes.

Dataset Processing: We compute the multi-level Steiner tree as described in Sec. 7. The details about the terminal selection method, number of levels, and desired edge lengths are provided in that section. We assign different edge lengths to compare our three algorithms. In this case, the edge lengths increases linearly as we go from the lowest level to top. Note that we can equivalently consider such a multi-level tree as a single-level tree with different edge lengths. Specifically, a multi-level tree has a hierarchical structure: all the nodes and edges of a particular level are also present in the lower levels. Hence, for each edge, we consider the highest level where the edge is present. We assign the desired edge length of this edge to the edge length of that level. With this in mind, we compare our three algorithms using a single-level tree, where all edges are present and have different desired edge lengths.

Since the sfdp+p and CIR cannot handle different edge lengths, to compare all algorithms we consider the setting where the trees are given as above, but the edge lengths are uniform.

Quantitative Evaluation: We measure the three optimization goals: desired edge length preservation, compactness, and runtime.

Desired Edge Length (DEL): evaluates the normalized desired edge lengths in each layer. Given the desired edge lengths $\{l_{ij} : (i,j) \in E\}$, defined in Sec. 7, and coordinates of the nodes X in the computed layout, we evaluate DEL with the following formula:

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

This measures the root mean square of the relative error, producing a positive number, with 0 corresponding to perfect realization.

Compactness Measure (CM): measures the ratio between the total 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). CM scores are in the range $[0,1]$, where 1 corresponds to perfect area utilization.

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

Results: We evaluate the performance of five algorithms on 7 graphs extracted from the 4 datasets above. The first two algorithms DELG and CG are only applied to

[2] http://tolweb.org/tree/

[3] https://genealogy.math.ndsu.nodak.edu/

			Desired Edge Length ↓					Compactness ↑					Runtime (sec) ↓						Crossings ↓	
	$\|V\|$	Edge Length	CIR	sfdp+p	DELG	CG	BT	CIR	sfdp+p	DELG	CG	BT	CIR	sfdp+p	DELG	CG	BT	BT*	sfdp+p	others
Last.FM	2,588	uniform	2.06	0.56	**0.19**	0.45	0.22	3e-7	0.04	0.01	**0.13**	0.05	**5**	8	926	309	54	15	147	**0**
		linear	-	-	**0.13**	0.45	0.30	-	-	0.01	**0.07**	0.01	-	-	1635	233	55	**15**	-	**0**
Topics	5,001	uniform	1.93	0.72	**0.29**	0.38	0.65	7e-6	0.03	0.01	**0.05**	0.02	44	**22**	3997	1000	175	32	203	**0**
		linear	-	-	**0.15**	0.30	0.35	-	-	0.01	**0.05**	0.01	-	-	8456	1096	178	**32**	-	**0**
Tree of Life	2,934	uniform	1.24	0.53	**0.43**	0.49	0.69	6e-6	**0.09**	0.01	**0.09**	0.07	**7**	7	3157	333	62	36	134	**0**
		linear	-	-	**0.45**	0.47	0.49	-	-	0.01	**0.10**	0.01	-	-	1178	698	61	**55**	-	**0**
Math Genealogy	3,016	uniform	3.15	0.78	**0.22**	0.40	0.50	3e-6	**0.13**	0.02	0.02	0.02	**5**	6	2067	774	57	14	1508	**0**
		linear	-	-	**0.31**	0.34	0.36	-	-	0.02	**0.03**	0.02	-	-	1340	563	64	**14**	-	**0**
Topics (large)	34,758	uniform	3.88	**0.76**	-	-	0.84	3e-6	**0.005**	-	-	0.001	1259	**258**	-	-	9346	2925	7713	**0**
Tree of Life (large)	35,960	uniform	2.17	**0.83**	-	-	1.29	3e-6	**0.006**	-	-	0.001	792	**455**	-	-	8780	2184	12088	**0**
Genealogy (large)	100,347	uniform	-	0.88	-	-	**0.83**	-	**0.007**	-	-	0.002	-	**2041**	-	-	-	8689	27478	**0**

Table 1: Quantitative algorithmic comparison using desired edge length preservation, compactness and runtime for 5 algorithms. The ↓ next to a criterion indicates lower scores are better and ↑ indices that higher scores are better.

the 4 small graphs. We show the layouts created by the five algorithms for the Last.fm graph in 5 for the uniform edge length setting. We provide other layouts in the supplementary material. These figures highlight some significant differences which stand out visually.

We provide the quantitative data in Table 1. From these results we can see that in the linear edge length setting DELG does best in the desired edge length measure, CG does best in the compactness measure, and BT* does best in runtime. This is exactly what one expects to see, given that each of these algorithms optimizes the corresponding feature. Note that we report two numbers for runtime of the parallel algorithm: BT (running on a laptop) and BT* (running on a server).

In the uniform edge length setting, DELG performs well in desired edge length preservation. Similarly, CG performs well in compactness, although sfdp+p outperforms it one instance (at the expense of edge crossings). The CIR method is the fastest, and the running times of sfdp+p and BT* are comparable.

For the 3 large graphs we only consider the uniform edge setting. The desired edge length scores of sfdp+p and BT are comparable with sfdp+p outperforming BT in 2 of the 3 cases. sfdp+p performs well both in compactness and running time, at the expense of many edge crossings; see last two column of Table 1.

Experimental Environment: We conducted most the experiments on a laptop, except one set of experiments which used a Skylake server machine (indicated by BT*). The laptop is configured with MacOS, 2.3 GHz Dual-Core Intel Core i5, 8GB RAM, and 4 logical cores. The server is configured with Linux OS, Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz, 256GB RAM, 2 sockets, and 24 cores per socket.

9. Discussion, Limitations, Conclusions

Comparing our three algorithms shows that they do well in their respective goals. Table 1 confirms that the DELG algorithm outperforms the others in desired edge length preservation, CG algorithm outperforms the others in compactness and BT is the fastest.

Comparing the three new algorithms to the two prior ones also seem encouraging, even though sfdp+p introduces crossings in all layouts and CIR has compactness scores that are orders of magnitude worse. On all 4 small graphs: DELG outperforms sfdp+p and CIR in edge lengths; CG outperforms CIR and sfdp+p in compactness in most cases; BT is slower than both sfdp+p and CIR, but not by much.

While we attempted to compare our readable tree layout framework with prior algorithms, we only used two. Similarly, we used only four real-world datasets and seven trees extracted from them for our evaluation. Further experiments with different types of trees, and with synthetically generated trees that test the limits of the prior and proposed methods (e.g., with respect to balance, degree distribution, diameter, etc.) are needed.

The utility of crossings-free, compact layouts that capture pre-specified edge lengths and show all node labels without overlaps needs to be evaluated. While intuitively these seem like highly desirable features (non-overlapping labels make for readable layouts, non-crossing layouts help grasp the underlying structure, compact layouts require less panning and zooming), a human subjects study can validate these goals.

DELG and CG do not scale well to trees with more than 5000 nodes. It was with this in mind that we added BT, which is similar to DELG but takes advantage of parallel computation and modern computer architectures. A similarly optimized version of CG would help optimize compactness for large trees. Better yet, a scalable algorithm not based on DELG or CG that guarantees o edge crossings and no label overlaps and fine-tunes one or more desirable properties is needed.

Even though "tree layout" is an old, well-known, and arguably solved problem, the "readable tree layout problem" shows that there is more work to be done in this domain. We propose an algorithmic framework for creating readable tree layouts and 3 algorithms that

guarantee crossings-free layouts with non-overlapping node labels. Each of the algorithms optimizes one desirable property: DEL, compactness, and runtime. The utility of such algorithms goes beyond drawings of trees, as illustrated by several examples and a video on the project website[4]. All source code, datasets and analysis will be made available upon acceptance.

References

[1] H.-J. Schulz, "Treevis. net: A tree visualization reference," *IEEE Computer Graphics and Applications*, vol. 31, no. 6, pp. 11–15, 2011.

[2] K. Börner, A. Maltese, R. N. Balliet, and J. Heimlich, "Investigating aspects of data visualization literacy using 20 information visualizations and 273 science museum visitors," *Information Visualization*, vol. 15, no. 3, pp. 198–213, 2016.

[3] E. M. Reingold and J. S. Tilford, "Tidier drawings of trees," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 2, pp. 223–228, 1981.

[4] P. Eades, "A heuristic for graph drawing," *Congressus Numerantium*, vol. 42, pp. 149–160, 1984.

[5] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and Experience*, vol. 21, no. 11, pp. 1129–1164, 1991.

[6] U. Brandes and C. Pich, "Eigensolver methods for progressive multidimensional scaling of large data," in *Graph Drawing*, pp. 42–53, Springer, Springer, 2007.

[7] Y. Koren, L. Carmel, and D. Harel, "Ace: A fast multiscale eigenvectors computation for drawing huge graphs," in *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, pp. 137–144, IEEE, 2002.

[8] R. Hadany and D. Harel, "A multi-scale algorithm for drawing graphs nicely," *Discrete Applied Mathematics*, vol. 113, no. 1, pp. 3–21, 2001.

[9] P. Gajer, M. Goodrich, and S. Kobourov, "A fast multi-dimensional algorithm for drawing large graphs," *Computational Geometry: Theory and Applications*, vol. 29, no. 1, pp. 3–18, 2004.

[10] K. W. Boyack, R. Klavans, and K. Börner, "Mapping the backbone of science," *Scientometrics*, vol. 64, no. 3, pp. 351–374, 2005.

[11] M. K. Rahman, M. H. Sujon, and A. Azad, "BatchLayout: A batch-parallel force-directed graph layout algorithm in shared memory," in *2020 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 16–25, IEEE, 2020.

[12] A. Arleo, W. Didimo, G. Liotta, and F. Montecchiani, "A distributed multilevel force-directed algorithm," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 754–765, 2018.

[13] J. Ellson, E. Gansner, L. Koutsofios, S. North, G. Woodhull, S. Description, and L. Technologies, "Graphviz — open source graph drawing tools," in *Lecture Notes in Computer Science*, pp. 483–484, Springer, 2001.

[14] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, and P. Mutzel, "The open graph drawing framework (OGDF)," *Handbook of Graph Drawing and Visualization*, pp. 543–569, 2011.

[4] `https://readabletreelayout.github.io/demo/overview.html`

[15] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," in *International AAAI Conference on Web and Social Media*, 2009.

[16] R. Wiese, M. Eiglsperger, and M. Kaufmann, "yfiles visualization and automatic layout of graphs," in *Graph Drawing Software*, pp. 173–191, Springer, 2004.

[17] C. Nobre, M. Meyer, M. Streit, and A. Lex, "The state of the art in visualizing multivariate networks," in *Computer Graphics Forum*, vol. 38, pp. 807–832, 2019.

[18] S. Hadlak, H. Schumann, and H.-J. Schulz, "A survey of multi-faceted graph visualization.," in *EuroVis (STARs)*, pp. 1–20, 2015.

[19] K. Marriott, P. Stuckey, V. Tam, and W. He, "Removing node overlapping in graph layout using constrained optimization," *Constraints*, vol. 8, no. 2, pp. 143–171, 2003.

[20] E. R. Gansner and S. C. North, "Improved force-directed layouts," in *Proceedings of the 6th International Symposium on Graph Drawing*, Graph Drawing '98, pp. 364–373, Springer, 1998.

[21] E. R. Gansner and Y. Hu, "Efficient node overlap removal using a proximity stress model," in *Graph Drawing* (I. G. Tollis and M. Patrignani, eds.), (Berlin, Heidelberg), pp. 206–217, Springer, 2009.

[22] L. Nachmanson, A. Nocaj, S. Bereg, L. Zhang, and A. E. Holroyd, "Node overlap removal by growing a tree," *J. Graph Algorithms Appl.*, vol. 21, no. 5, pp. 857–872, 2017.

[23] M. Luboschik, H. Schumann, and H. Cords, "Particle-based labeling: Fast point-feature labeling without obscuring other visual features," *IEEE transactions on visualization and computer graphics*, vol. 14, no. 6, pp. 1237–1244, 2008.

[24] K. Mote, "Fast point-feature label placement for dynamic visualizations," *Information Visualization*, vol. 6, no. 4, pp. 249–260, 2007.

[25] C. Kittivorawang, D. Moritz, K. Wongsuphasawat, and J. Heer, "Fast and flexible overlap detection for chart labeling with occupancy bitmap," in *IEEE VIS Short Papers*, 2020.

[26] Y. Hu, "Efficient, high-quality force-directed graph drawing," *Mathematica Journal*, vol. 10, no. 1, pp. 37–71, 2005.

[27] C. Bachmaier, U. Brandes, and B. Schlieper, "Drawing phylogenetic trees," in *International Symposium on Algorithms and Computation*, pp. 1110–1121, 2005.

[28] K. Rahman, M. Sujon, and A. Azad, "Force2Vec: Parallel force-directed graph embedding," in *Intl. Conference on Data Mining (ICDM)*, pp. 442–451, IEEE, 2020.

[29] R. Ahmed, P. Angelini, F. Sahneh, A. Efrat, D. Glickenstein, M. Gronemann, N. Heinsohn, S. Kobourov, R. Spence, J. Watkins, and A. Wolff, "Multi-level Steiner trees," *ACM J. Exp. Algorithmics*, vol. 24, no. 1, pp. 2.5:1–2.5:22, 2019.

[30] E. R. Gansner, Y. Hu, and S. Kobourov, "Gmap: Visualizing graphs and clusters as maps," in *2010 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 201–208, 2010.

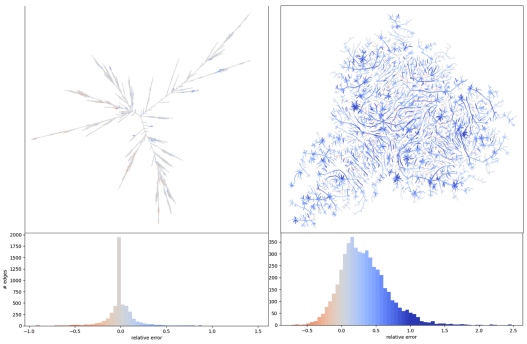
[31] R. Burd, K. Espy, I. Hossain, S. Kobourov, N. Merchant, and H. Purchase, "GRAM: Global research activity map," in *Intl. Conference on Advanced Visual Interfaces (AVI)*, pp. 31:1–31:9, ACM, 2018.

A. Supplementary Material

In this section, we provide some supplementary materials. First, we include a qualitative analysis of the algorithms, this compares the DELG and CG and includes a brief discussion on the scalability. Then, we include some further details of our different algorithms and illustrate them with pictures. We then provide some augmented images of our algorithms on different datasets that show a zoomed-in view of the images.

A.1. Qualitative Analysis

Here we take a closer look at the results returned by the three new algorithms and the two prior ones. Fig. 5 shows the results of last.FM graphs. As the name implies, circular layout (CIR) wraps branches of the tree into spiraling circles to form a compact layout. Edges close to the center of the tree are stretched in order to provide large areas for the subtrees, resulting in poor edge length realization. Since leaves are drawn in small regions, the overall layout must be scaled a lot in order to shown the labels without overlaps, yielding poor compactness. The sfdp+p results are consistently good in compactness, at the expense of many crossings. Our three new methods are better at capturing the global structure. DELG is best at realizing edge lengths, at the expense of compactness. CG is best at area utilization, at the expense of desired edge lengths. BT is the fastest of the three, although it performs worse in both desired edge lengths and area utilization.



(a) DELG (b) CG

Figure 6: Analysis of failure modes in desired edge length preservation. **Top left:** In the layout of the topic graph computed by DELG, most edges are drawn with their desired edge lengths. **Bottom left:** A histogram of relative errors in desired edge length preservation, colored in the same way as the layout. **Right**: The same analysis on the CG algorithm, showing CG stretches (blue) more edges to improve compactness.

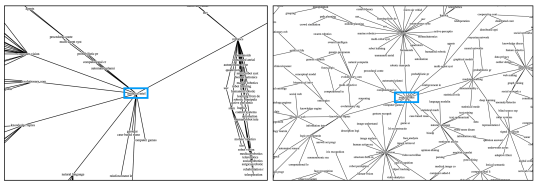


Figure 7: Analysis of failure modes in layout compactness. **Left:** The topic graph layout returned by DELG has more unused space and fails to distribute heavy subtrees around the 'Artificial Intelligence' node (pointed and circled in blue) evenly. **Right:** CG generates a more balanced layout and is able to distribute subtrees more evenly.

Next, we look more closely at the layouts obtained from DELG and CG. First, we focus on desired edge length preservation. Figure 6 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 $(u,v) \in E$, it measures the discrepancy between the actual edge length $||X_u - X_v||$ in the drawing and the given desired edge length l_{uv}: relative_error(u,v) $= (||X_u - X_v|| - l_{uv})/l_{uv}$.

With DELG, we observed from the left layout and histogram in Figure 6 that most edges are drawn with their desired edge lengths due to its edge-length guided initialization. We can see a few stretched or compressed edges (colored in blue and red in Figure 6), 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. The errors are larger in CG, which seems to be due to the way label overlaps are handled: in dense regions, e.g. around high degree nodes, the edges are more likely to be stretched, whereas on the periphery the edge lengths are better preserved.

Next we look at the compactness of the two 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 7. From the layout overview in 7, we can already see that DELG uses space less efficiently, it has more empty, white space. The 'Artificial Intelligence' node, highlighted in blue in 7, is close to multiple heavy subtrees such as those from the nodes 'natural language processing', 'machine learning', and 'computer vision'. CG distributes the heavy trees evenly, due to its initial sfdp layout. DELG, on the other hand, places most heavy branches on the left side, resulting in a less efficient use of drawing area.

Scalability: We experimented further with the BT algorithm to evaluate how it behaves with larger number of cores and with larger number of nodes, shown in Fig. 8(a) and Fig. 8(b), respectively. In Fig. 8(a), we

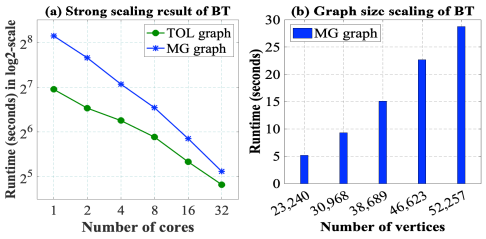


Figure 8: (a) Strong scaling results of Tree of Life (TOL) dataset (35,960 nodes) and Math Genealogy (MG) dataset (52,257 nodes). (b) Graph scaling results for different size of MG trees using 48 cores. Only per-iteration runtime is reported.

show strong scaling results for the Tree of Life and Math Genealogy datasets. For both datasets, we observe that the runtime decreases almost linearly as the number of cores increases. In Fig. 8(b), we report the per-iteration runtime on the Math Genealogy tree when increasing the size of the trees. We observe that the runtime increases almost linearly with the size of the tree. This provides support for the scalability of the BT. It is notable that while BT does not outperform DELG in desired edge lengths or CG in compactness, BT's numbers are within a small constant factor of the best values.

B. DELG crossings-free initialization details

We illustrate the process by an example, see Figure 9. A symbolic representation of the input tree is shown in Figure 9a. Here the root node r has three children v_1, v_2 and v_3. The subtree rooted at v_i is denoted by S_i. The number of nodes in the subtrees S_1 and S_3 are equal and larger than the number of nodes in S_2. Algorithm **??** places the root vertex r and divides the space into three regions p_1rp_2, p_2rp_3 and p_1rp_3, see Figure 9b. Here we denote the angular region created by two segments p_1p_2 and p_1p_3 by $p_2p_1p_3$. The regions p_1rp_2, p_2rp_3 and p_1rp_3 are assigned to the nodes v_1, v_2 and v_3 respectively. The region p_2rp_3 is smaller than the other two regions because Algorithm **??** divides the regions proportionally to the size of the subtrees, and S_2 is smaller in size compared to the other two subtrees. Algorithm **??** now places the vertices in their regions. Consider the vertex v_1, the region assigned to v_1 is p_1rp_2. The vertex v_1 is placed on the bisector of rp_1 and rp_2 in such a way that the distance of rv_1 is equal to the desired edge length of the edge (r, v_1). Similarly, the algorithm places v_2 and v_3. The algorithm now translates the angular regions to the corresponding child vertices. For example, consider the child vertex v_2. We translate the line rp_3 to v_2p_6 and rp_2 to v_2p_7. The angular region for the subtree S_2 is $p_6v_2p_7$.

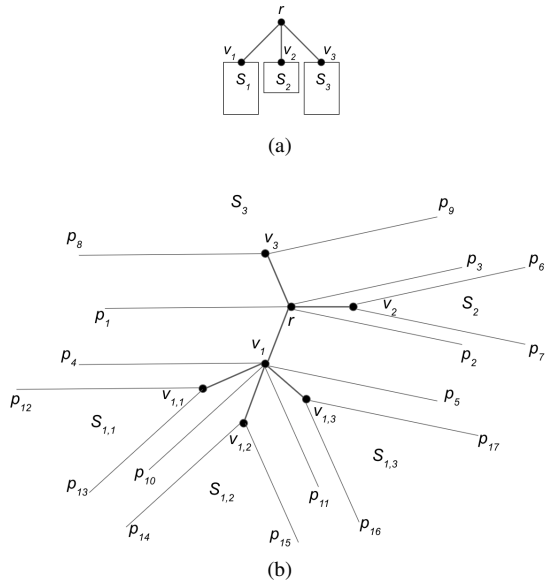


Figure 9: Illustrating different steps of Algorithm **??**. The input graph is shown in Figure 9a. The division of angular regions based on the size of the subtrees is shown in Figure 9b.

Similarly, the regions of S_1 and S_3 are $p_4v_1p_5$ and $p_8v_3p_9$ respectively. Note that the region is unbounded, and one can draw edges with arbitrary length in a region without introducing any crossing with the edges of other regions. The algorithm continues this process recursively. For example, we assume that v_1 has three children $v_{1,1}, v_{1,2}$ and $v_{1,3}$. For simplicity we assume that each child subtree has an equal number of edges. The algorithm divides $p_4v_1p_5$ into three equal sized regions $p_4v_1p_{10}, p_{10}v_1p_{11}$ and $p_{11}v_1p_5$. We then place the children vertices in the corresponding regions and translate the regions. When the input is a balanced tree this algorithm computes a symmetric layout; see 10.

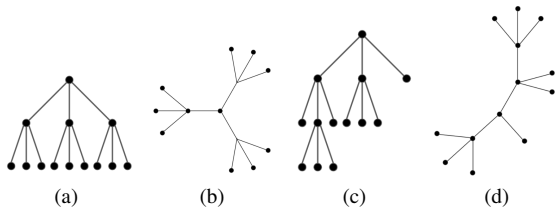


Figure 10: Illustrating the output layout of **??** w.r.t. different types of inputs. If the tree is balanced (see 10a) then the output is symmetric (see 10b). If the tree is not balanced (see 10c) then the output is not symmetric (see 10d).

This initial layout is crossing free and preserves desired edge lengths. However, it may contain label overlaps. For example, in 11 we show a layout of a large real-world graph which contains many dense regions that will cause label overlaps. To remove these overlaps, we start with this initial layout as an input and apply a customized force-directed algorithm as we mentioned in the main paper. The force-directed algorithm partially removes label overlaps. We run a final overlap removal process to completely remove all overlaps.

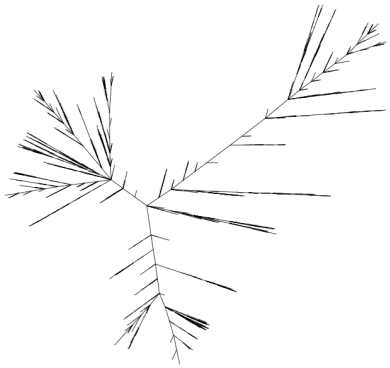


Figure 11: A layout of **??** that preserves desired edge lengths.

C. DELG Final Overlap Removal Details

The second step of the algorithm does not necessarily remove all label overlaps. This is due to the the force-directed algorithm containing two different types of forces: collision force and edge force. In some cases, one force may act opposite to the other force., see Figure 12. For every remaining label overlap we run a post-processing step. 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 bounding box that has $2-3\%$ of the drawing area. We denote the set of nodes in that bounding box by V'. We then sample 100 random points from that bounding box. For each of the 100 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. Note that this process does not guarantee that we will get rid of all label overlaps. However, we found this process very effective in practice and running it $3-4$ times has given an overlap-free drawing for all datasets.

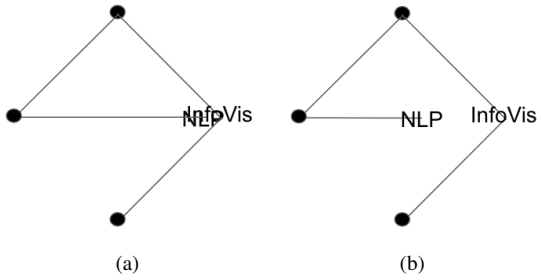


Figure 12: The corresponding edge of "NLP" has a larger desired edge length. Hence, due to edge force, the algorithm provides label overlaps as shown in Figure 12a. By applying post-processing we can remove such overlap as shown in Figure 12b.

D. The CG Algorithm Details

The DELG algorithm focuses on preserving desired edge lengths. On the other hand, the CG algorithm generates a more compact layout. To achieve better compactness it applies additional forces and also transforms the coordinates. 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; see Fig. 13.

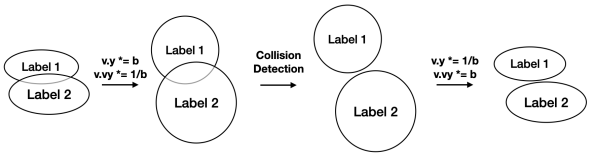
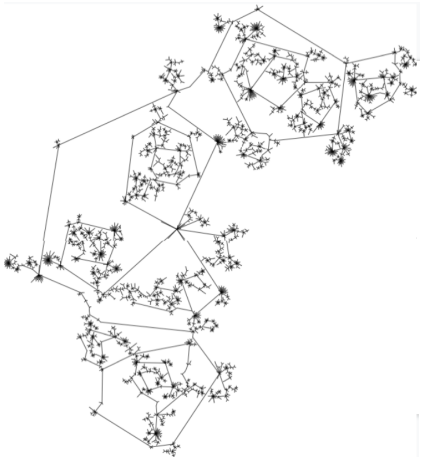


Figure 13: Illustration of Algorithm **??**: label overlap removal in the CG algorithm using oval collision detection.

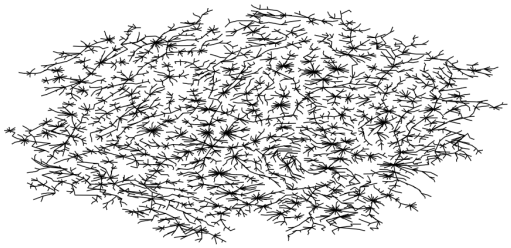
E. Evaluation

We now provide some images that show the outputs from different algorithms on different graphs. In 14, we give layouts of uniform Topics Graph computed by two existing algorithms (CIR and sfdp+p) and ours (DELG, CG and BT). In 15, we compare the layout of the Last.FM graph generated using DELG, CG and BT. The DELG and BT focus more on preserving desired edge length which helps to capture the overall topology of the layout. However, if we zoom in then we can see that the drawing

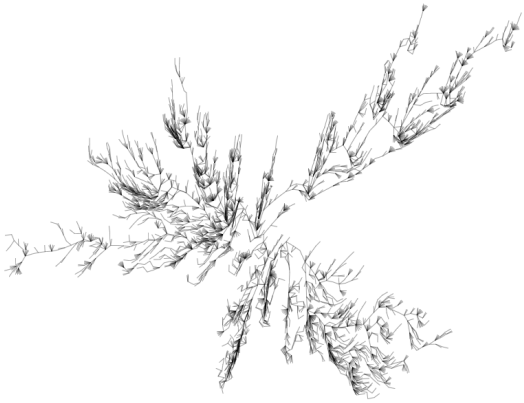
is not compact: there are some free spaces among the labels. On the other hand, the CG algorithm focuses more on compactness and by zooming in the layout shows that relatively more labels are drawn compactly. Similarly, the DELG and BT algorithms preserve the desired edge lengths while CG optimizes compactness for topics and tree of life graphs, see 16 and 17 respectively.



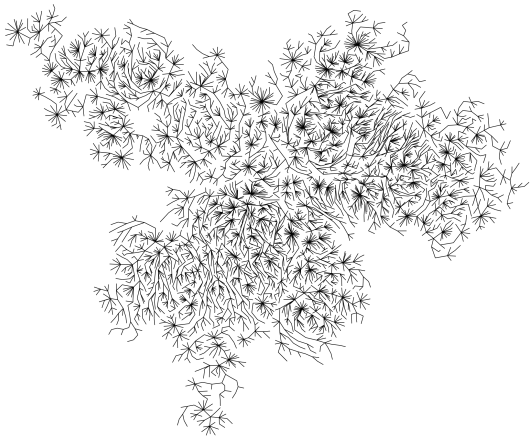
(a) CIR



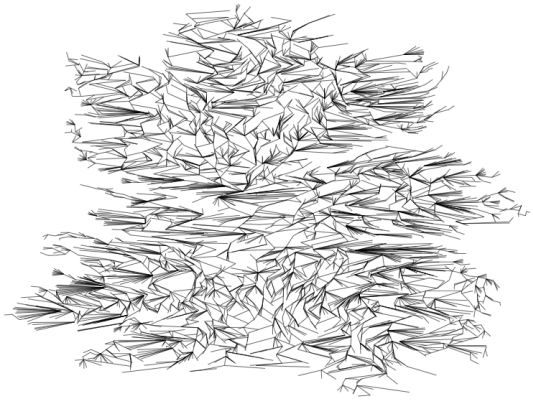
(b) sfdp+p



(c) DELG

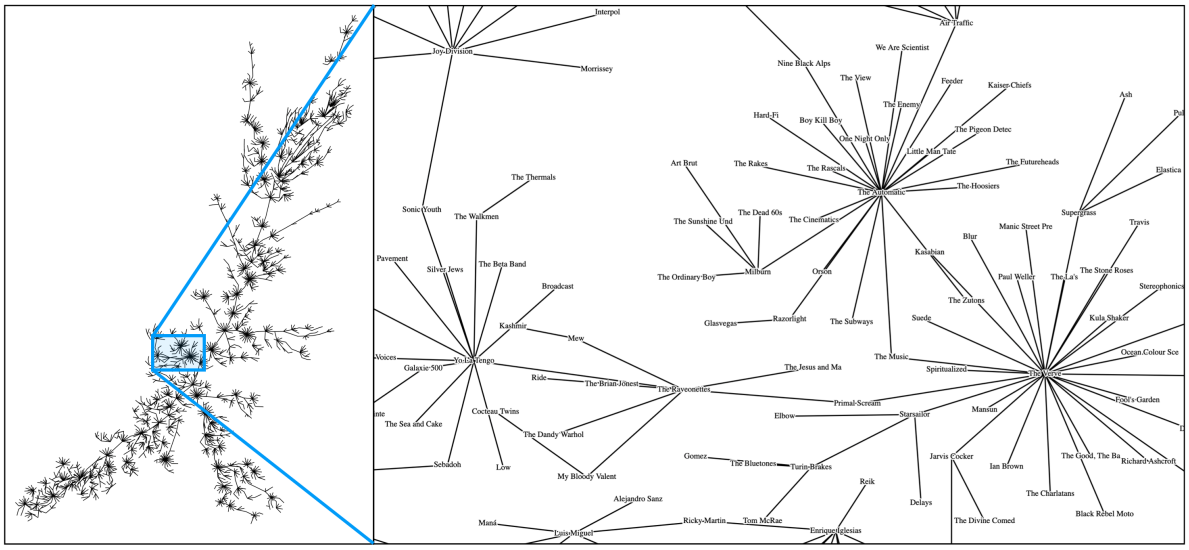


(d) CG

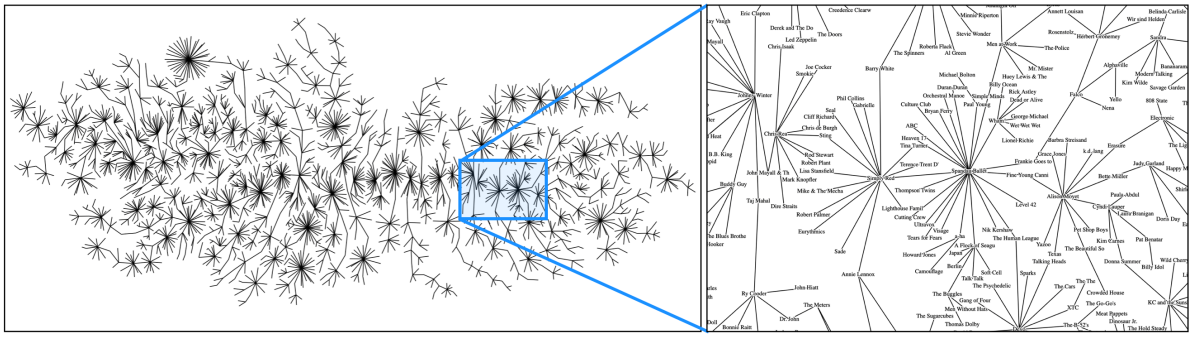


(e) BT

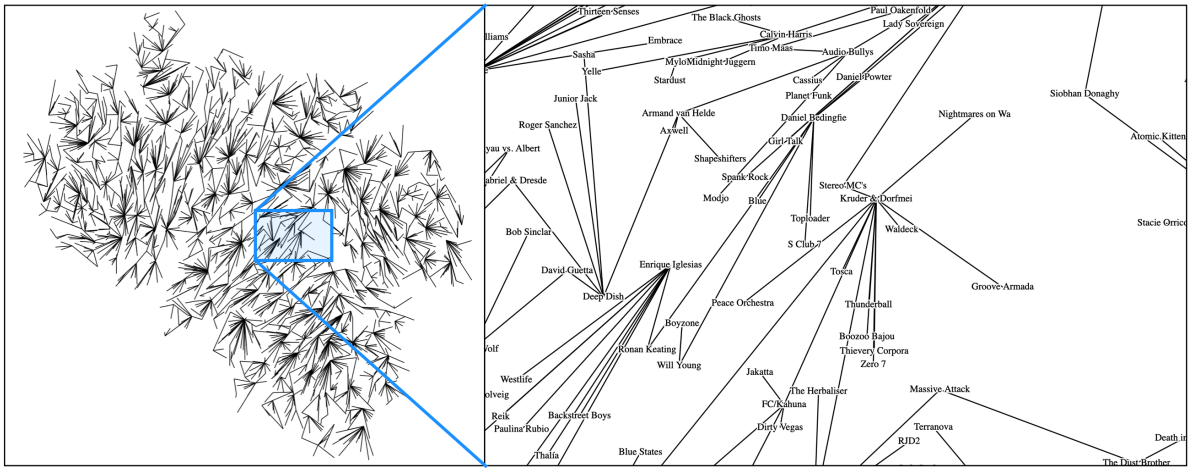
Figure 14: Comparison of the tree layout structure of the uniform Google Topics graph drawn with CIR, sfdp+p, DELG, CG, and BT.



(a) DELG

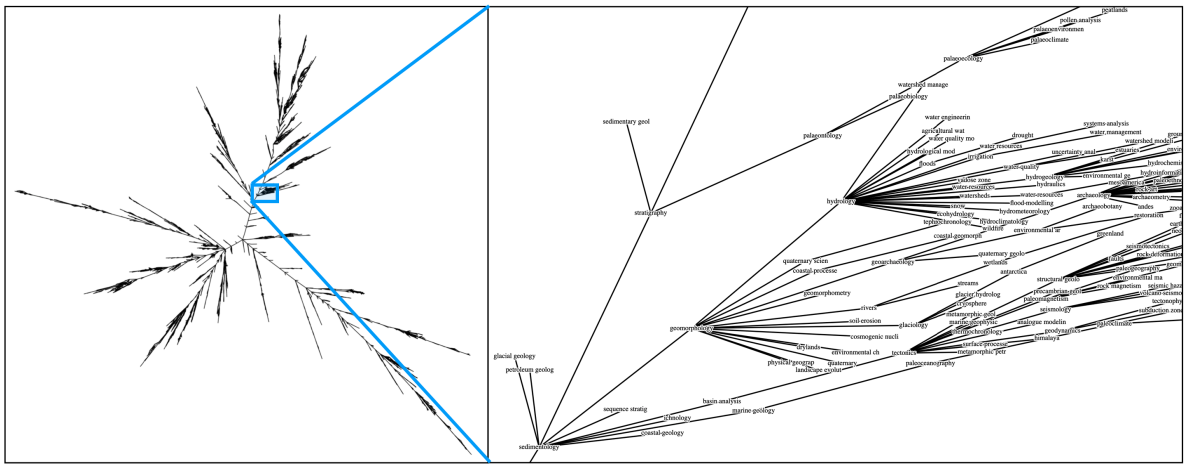


(b) CG

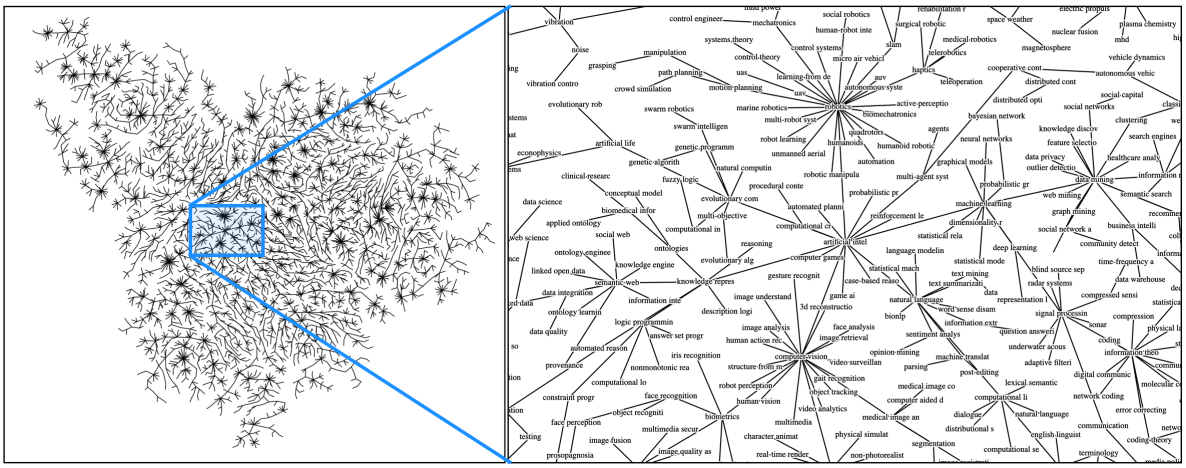


(c) BT

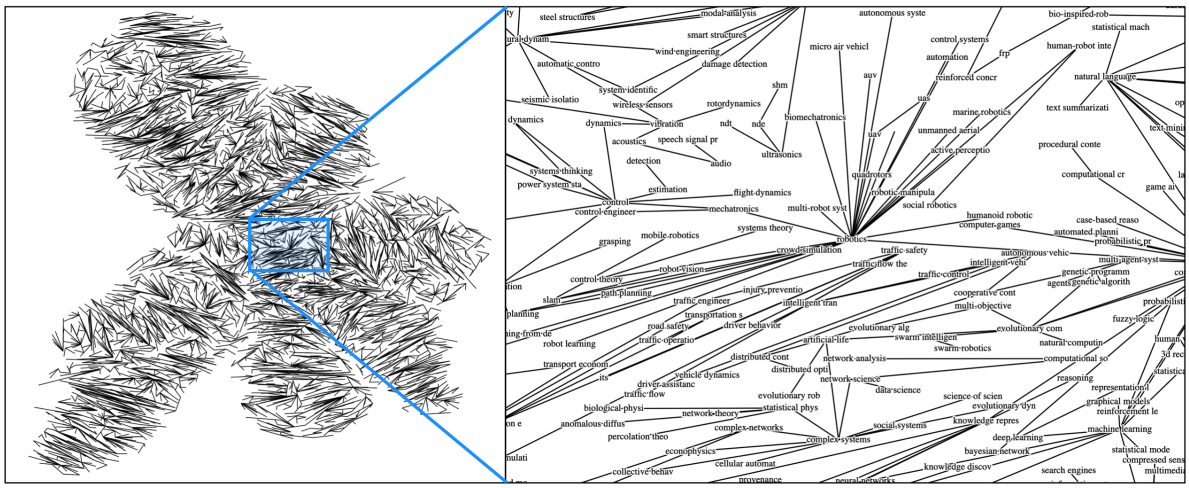
Figure 15: Comparing Last.FM linear layouts drawn with our algorithms.



(a) DELG

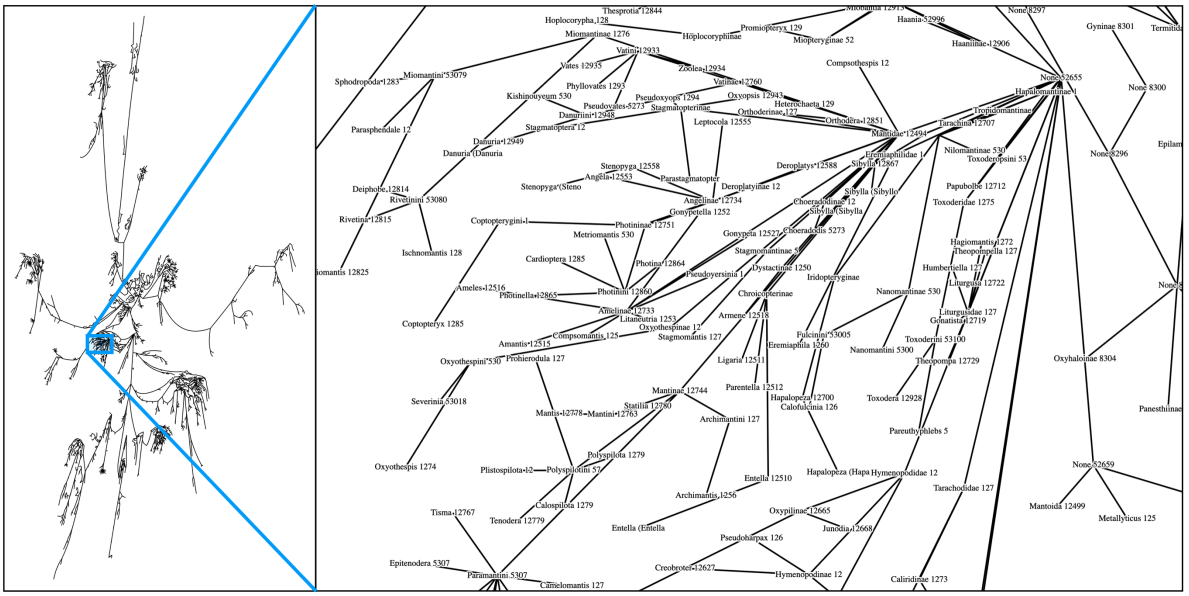


(b) CG

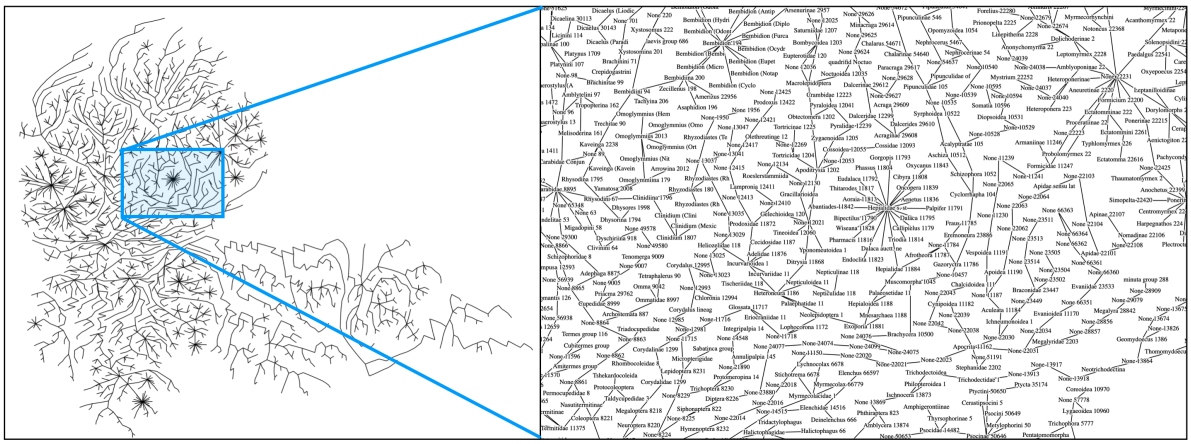


(c) BT

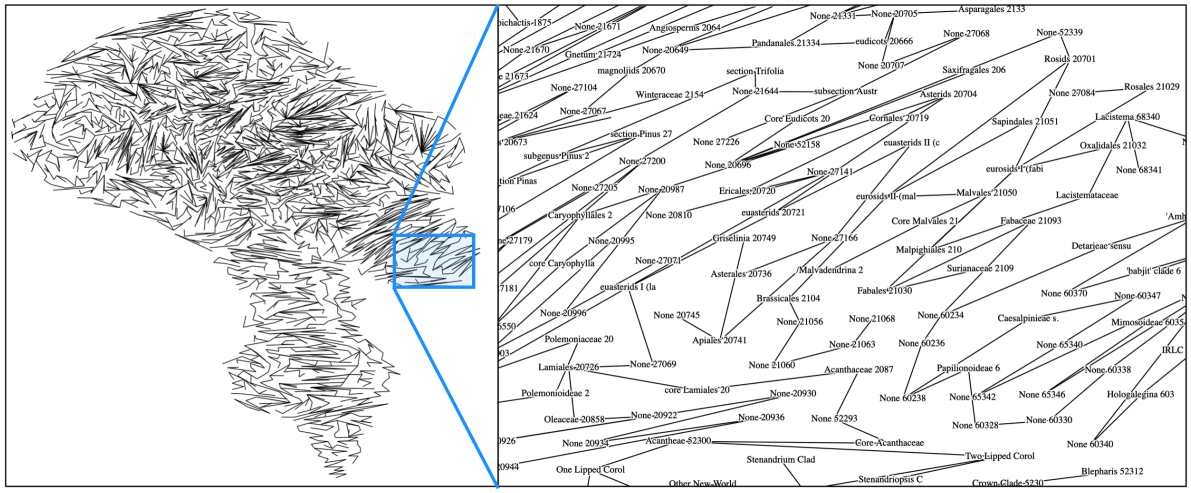
Figure 16: Comparing Topics linear layouts drawn with our algorithms.



(a) DELG



(b) CG



(c) BT

Figure 17: Comparing Tree of Life linear layouts drawn with our algorithms.