This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TVCG.2018.2865265

Temporal Treemaps: Static Visualization of Evolving Trees

Wiebke Köpp and Tino Weinkauf

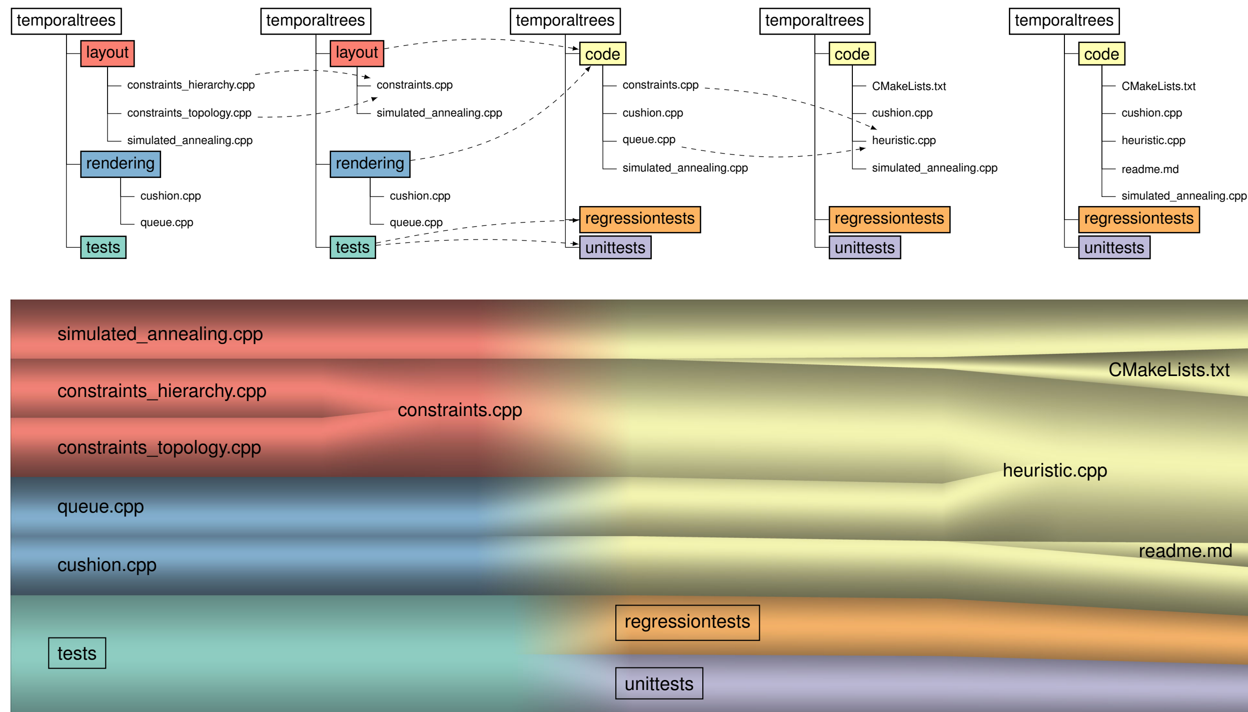


Figure 1. Our method optimizes a layout for a series of trees and renders it using an adaptation of cushion rendering. Notably, we support topological changes to the trees such as merges and splits on all hierarchy levels. The shown example displays five time steps in the evolution of a file system hierarchy, roughly resembling the code base for this paper, where folders and files merge, split, appear, and disappear (top). Our method displays these evolving trees in a single layout paying attention to both the merges/splits as well as the hierarchical nesting (bottom).

Abstract— We consider temporally evolving trees with changing topology and data: tree nodes may persist for a time range, merge or split, and the associated data may change. Essentially, one can think of this as a time series of trees with a node correspondence per hierarchy level between consecutive time steps. Existing visualization approaches for such data include animated 2D treemaps, where the dynamically changing layout makes it difficult to observe the data in its entirety. We present a method to visualize this dynamic data in a static, nested, and space-filling visualization. This is based on two major contributions: First, the layout constitutes a graph drawing problem. We approach it for the entire time span at once using a combination of a heuristic and simulated annealing. Second, we propose a rendering that emphasizes the hierarchy through an adaption of the classic cushion treemaps. We showcase the wide range of applicability using data from feature tracking in time-dependent scalar fields, evolution of file system hierarchies, and world population.

Index Terms—Treemaps, Temporal trees.

1 Introduction

Hierarchical data structures are common. Filesystems, source code repositories, HTML/XML files, or the organizational structure of states and companies are only few examples. The large amount of different visualization approaches for trees speaks to both the importance of the data type and its ability to spark the interest of the visualization community.

• *All authors are with KTH Royal Institute of Technology, Stockholm, Sweden. E-mail: {wiebkek | weinkauf}@kth.se .*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

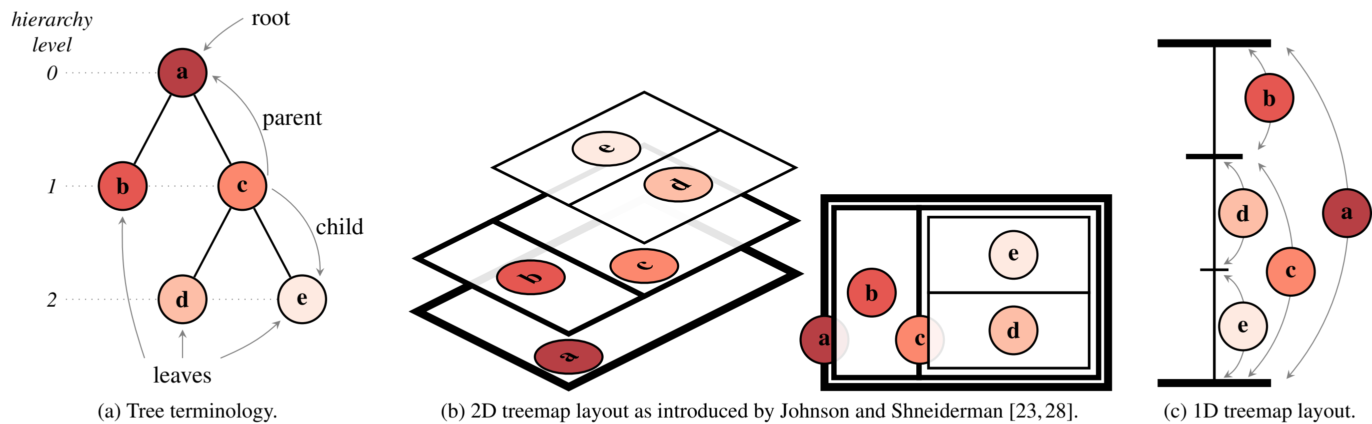
It is particularly useful to associate data to the nodes of a tree. Different methods can then be used to reveal the largest or smallest parts of the hierarchy. The concept of nested layouts, in particular treemaps [28], is a wide-spread visualization approach for such trees.

Data associated with a tree may change over time and it is of high interest to observe the temporal development of this data. Dynamically adjusting treemaps [19,30–33,37] have been proposed for this. Furthermore, stream graphs enhanced with hierarchical information [3,14,40] provide a static overview of the entire temporal development of trees with dynamically changing data.

This paper deals with trees whose data and topology change over time. Essentially, the entire tree can transform almost arbitrarily as long as it remains a tree. To be precise, we consider a time series of trees with a node correspondence per hierarchy level between consecutive time steps. Tree nodes may appear and disappear, merge and split on all

Copyright (c) 2018 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TVCG.2018.2865265



(a) Tree terminology. (b) 2D treemap layout as introduced by Johnson and Shneiderman [23,28]. (c) 1D treemap layout.

Figure 2. Treemaps can be defined in different dimensions. A classic 2D layout makes excellent use of a 2D display space, but incorporating a temporal dimension is difficult. A 1D treemap layout frees up one spatial dimension for incorporating the temporal evolution of the tree.

hierarchy levels, and the associated data may change. We only exclude cases where a child changes its parent.

Temporally evolving trees can of course be visualized generically by applying any tree visualization method to each individual time step. This is often not satisfying, since discontinuous layout changes make it difficult to observe the data. Recently, Lukasczyk et al. [26] presented *Nested Tracking Graphs*, the first method to address such data. While the method has been described in the context of hierarchically nested graphs, such data can just as well be seen as a time series of trees. Lukasczyk et al. [26] represent time statically along a spatial axis and draw nodes as bands of varying thickness along this axis. They are nested inside their parent's band. The method exhibits a large amount of intersections, since the layout algorithm largely ignores the hierarchical relationships. Furthermore, the method has been designed for data where the parent's data value exceeds the sum of the children's data. This makes it inapplicable in the large number of application scenarios where the sum of the children's data equals the parent's data (e.g., a filesystem).

We give the following contributions:

- We present a novel layout algorithm for temporally evolving trees with changing topology and data. It considers the entire hierarchy and time span at once to produce a layout with as few intersections as possible. This is based on a combination of a heuristic and a simulated annealing optimization approach. The layout algorithm runs on the order of a few seconds.
- We propose a rendering scheme that emphasizes the hierarchy through an adaptation of the classic cushion treemaps [38]. This makes our method applicable to data sets where the sum of the children's data equals the parent's data.
- We propose a data structure for temporally evolving trees, which records only the changes to the tree. Besides its space-efficiency, it also reduces the computation times of the layout algorithm drastically.
- We showcase the wide range of applicability using data from feature tracking in time-dependent scalar fields, evolution of file system hierarchies, and world population.

The paper is organized as follows: Section 2 recollects the theory and previous work around trees, treemaps, and related visualization methods. We present our data structure for temporally evolving trees in Section 3. Our two main contributions, the layout and the rendering, are presented in Section 4. We evaluate our method in Section 5, show results from different domains in Section 6, and conclude in Section 7.

2 Related Work and Background

2.1 Trees and Treemaps

A tree $T = (N, E)$ is a hierarchical data structure with a set of nodes N and a set of directed edges E. Exactly one node is the *root* of the tree which has only outgoing edges. All other nodes have exactly one incoming edge from their *parent*, and zero or more outgoing edges to their *children.* Nodes with no children are called *leaves*. The *hierarchy level* of a node is determined as the number of edges in the uniquely defined path from the root to that node. See Figure 2a.

A common and useful way of associating data to a tree is to prescribe data values at the leaves and describe the data at each parent node $\mathbf{p}$ as the sum of the data of its children $\mathbf{c}_i$

$$d(\mathbf{p}) = \sum d(\mathbf{c}_i) \; . \tag{1}$$

A fitting example is a file system hierarchy where the files (leaves) occupy a certain size on disk, while the directories (parents) are a mere container with its size being the sum of their files and subdirectories.

Treemaps are a common tool for visualizing trees with associated data in a space-filling, nested layout. Each node is drawn in a size that relates to its data and serves as a container for drawing its children. Johnson and Shneiderman [23, 28] introduced the general concept: a rectangular space is associated with the root. Its children are drawn by dividing the rectangle along the x-axis in relation to the amount of data per child. This process continues by alternating the partitioning axis in each hierarchy level. Figure 2b illustrates this.

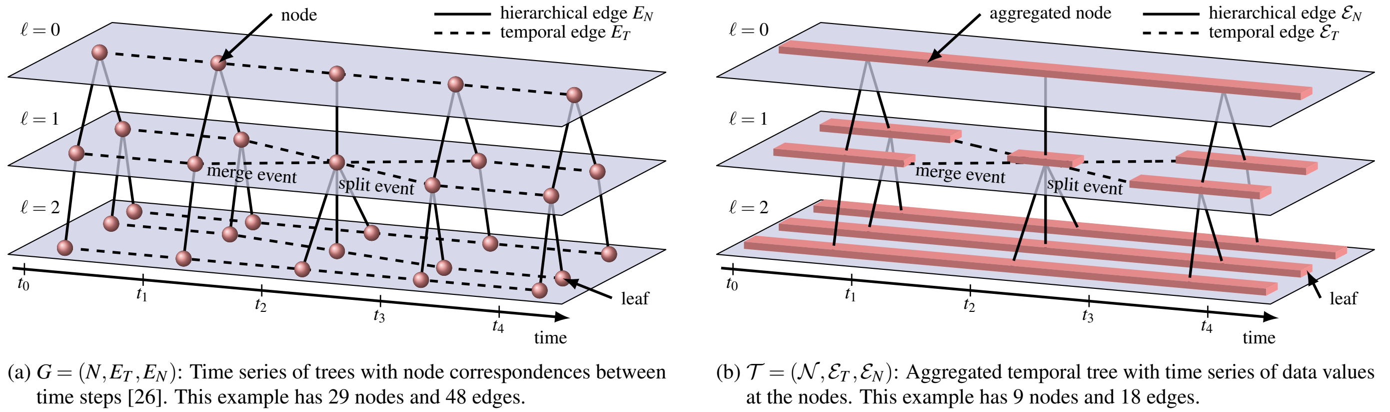
Different treemap layout variations have been proposed. Bruls et al. [8] strive for creating squares when partitioning the space in order to make tree nodes easier to compare. Bederson et al. [29] propose treemaps that preserve a given order in the original data. Other shapes than nested rectangles have been proposed as well, including general polygons within rectangles [16] , Voronoi cells [1], and bubbles [18] with the latter specifically targeting visualizing uncertainty in the data.

Treemap rendering methods differ in how a node's hierarchy level is displayed. Johnson and Shneiderman [23, 28] use boundary lines between the rectangles. Balzer and Deussen [1] emphasize the hierarchy through boundaries and color variation. Van Wijk and van de Wetering [38] propose *cushion treemaps*, which emulate a diffusely lit surface of varying height according to the hierarchy. In this paper, we adapt this strategy for our temporal treemaps, see Section 4.3.

Note that treemaps can also be defined in other dimensions. As noted already by Shneiderman [28], a treemap can be defined in 3D by subdividing a cube or in 1D by recursively subdividing a straight line. Neumann et al. [27] make use of a 1D nested layout to augment the hierarchical information with non-hierarchical relations. A 1D treemap frees up one display dimension for other information; we will exploit this later for incorporating the time dimension. Figure 2c shows a 1D treemap.

Copyright (c) 2018 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TVCG.2018.2865265



(a) $G = (N, E_T, E_N)$: Time series of trees with node correspondences between time steps [26]. This example has 29 nodes and 48 edges.

(b) $\mathcal{T} = (\mathcal{N}, \mathcal{E}_T, \mathcal{E}_N)$: Aggregated temporal tree with time series of data values at the nodes. This example has 9 nodes and 18 edges.

Figure 3. Aggregated temporal trees (right) encode information more space-efficiently than a time series of trees (left), especially when only a few nodes change their data values in every time step such as in the history of a file system. See also Section 5.

2.2 Visualization Approaches for Time-dependent Trees

Time can affect different aspects of a tree: The data at the nodes may change over time, or the topology of the tree may change, or both. Different visualization approaches cover different aspects, as discussed in the following.

Consider time-dependent data values on an otherwise static tree. This can be described using a time series at each node $[d_1, \ldots, d_n]$ for n time steps. In the case that the data value of a parent is the sum of its children, it suffices to store the time series at the leaves and restore them at all other nodes using Equation (1).

Many methods exist to represent several 1D graphs stacked on top of each other. Each individual graph is essentially a thick band representing the evolution of a topic or similar. *ThemeRiver* by Havre et al. [21] is among the first ones. The bands are ordered so that the evolution of a single band has as little as possible effect on the other bands. Detailed discussions of the amount of "wiggles" in such visualizations have been made by Byron and Wattenberg [11] as well as Bartolomeo and Hu [2]. A number of works [3, 14, 40] incorporate hierarchical information into stacked graphs. Hierarchy is conveyed through color encoding [14, 40], joint displaying of a tree [14, 40], or showing hierarchy layers separately [3] and facilitates interactive exploration of the data [3, 40]. In contrast to our approach, the topology of the tree remains static and nodes do not appear or disappear.

Several approaches extend the treemap concept to show time-dependent data on topologically static trees: animation is used to blend between the different time steps. The challenge is to balance between a good treemap layout and the number of layout changes [19, 30–33, 37].

Consider both the topology of the tree *and* the node data change over time. Recently, Lukasczyk et al. [26] presented a method for dealing with this kind of data. While it is phrased orthogonally as a method to work on nested graphs rather than a series of trees, it can just as well be seen as the latter. The method uses the Graphviz library [17] for the layout and while this produces great results, the final compositing step induces a larger number of intersections/crossings, because the hierarchical information is not used when computing the graph layout. We will detail this in Section 4.1. This paper presents a novel algorithm for the graph layout which is able to produce layouts with less intersections/crossings. Merges and splits in connection to hierarchy have also been explored by Cui et al. [15]. They visualize the evolution of topics in text corpora. However, not the entire tree is shown in each time step, but rather just a cut through the tree at possibly varying hierarchy levels.

Burch et al. [10] draw dynamically changing graphs with a hierarchical nesting next to each other. The resulting edge crossings are anticipated and managed with a splatting technique. Other methods for general dynamic graph visualization are discussed in Beck et al. [4].

2.3 Approaches to Constrained Ordering

We model our layout computation by creating a constrained ordering of objects. Each constraint requires the involved objects to appear consecutively within the ordering. A similar problem occurs in finding *path supports* in hypergraphs [9]. In contrast to regular graphs, edges of hypergraphs can involve more than two nodes. In a path support, all nodes incident to an edge are positioned next to each other. Algorithms exist to compute path supports in polynomial time iff all constraints can be fulfilled [6, 22]. However, computing optimal partial solutions, i.e., fulfilling as many constraints as possible if it is impossible to fulfill all constraints, has been shown to be NP-complete [20].

By interpreting leaves as nodes and constraints as edges of a hypergraph, our layout problem can be transformed into a path support problem. However, this neglects the temporal aspect of our nodes which leads to a substantially larger solution space. Nevertheless, future versions of our framework may potentially test for the existence of an optimal solution.

Apart from our scenario, constrained ordering also occurs in the evolution of stories [25, 36], where the co-location of characters imposes restrictions. The problem has been approached using a hypergraph formulation as above [36] as well as the computation of an initial solution allowing for interactive reordering to further minimize crossings [25].

3 Data Structure for Time-Dependent Trees

A practically useful definition for temporally evolving trees has recently been given by Lukasczyk et al. [26]: a *nested tracking graph* $G = (N, E_T, E_N)$ consists of a set of nodes $\mathbf{n}_t^\ell \in N$ where each node has a time step t, a hierarchy level ℓ, and a data value d. Edges in E_N exclusively describe the hierarchical relationships in a time step, i.e., the restriction of G to a time step t yields a proper tree $G|_t = (N|_t, E_N|_t)$ as defined above, or possibly a forest of such trees. Edges in E_T exclusively connect nodes of the same hierarchical layer ℓ, i.e., a restriction $G|_\ell = (N|_\ell, E_T|_\ell)$ gives a tracking graph, in which nodes may appear, disappear, merge, or split. Figure 3a shows an illustration.

Essentially, nested tracking graphs store a tree per time step and the tracking information in between. This definition serves well when tracking super- or sublevel sets [26] in time-dependent scalar fields, where every node's data value changes in every time step. However, such a data structure is rather space-consuming when dealing with data sets where only few nodes change in every time step such as in a file system hierarchy: storing an entire file system tree for each change to a file quickly leads to a large number of nodes and edges.

We define an *aggregated temporal tree* $\mathcal{T} = (\mathcal{N}, \mathcal{E}_T, \mathcal{E}_N)$ following the nested tracking graphs of Lukasczyk et al. [26] with the adaption that a node $\mathbf{n}_{t_a,t_b}^\ell \in \mathcal{N}$ exists over a time span $[t_a, t_b]$ and stores a time series of data values $[d_{t_a}, \ldots, d_{t_b}]$. This allows for a compact encoding of data values that records only changes. Figure 3b shows an illustration.

Copyright (c) 2018 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TVCG.2018.2865265

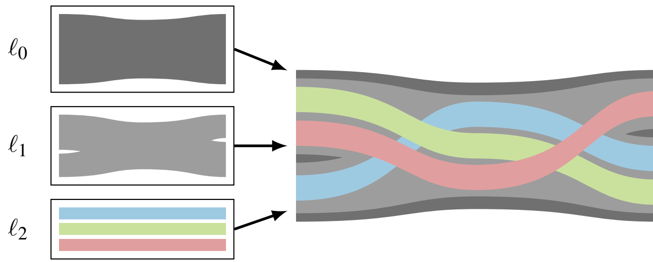


Figure 4. Lukasczyk et al. [26] uses Graphviz [17] to produce the graph layout in each hierarchy level independently (left). These layouts are optimized to have as few intersections as possible. But when combining all levels into a nested drawing by forcing the children to live inside their parents' space, it becomes apparent that the graph layout has no knowledge of the hierarchical nesting in G, and we get a number of intersections (right). This example has been created with the original software of [26]. Note that the underlying cause for the intersections is the violation of two hierarchical constraints in the sorting of the leaves in layer ℓ_2, as discussed in Section 4.2 and illustrated in Figure 6. See Figure 7 for the result of our method, and Figure 3 for the underlying data.

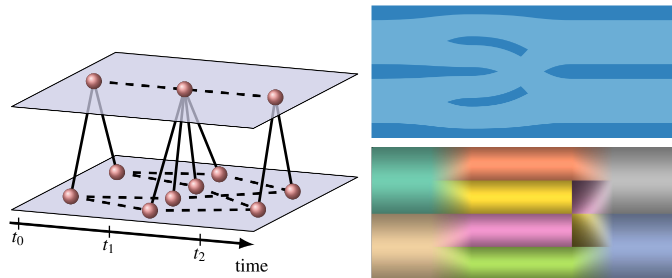


Figure 5. It is not always possible to draw a graph intersection-free. In particular, this is the case in our setting, where the nodes are aligned with time steps. The rendering method of Lukasczyk et al. [26] shows intersections directly (top-right image), whereas they appear as discontinuities in the cushion rendering (bottom-right image) discussed in Section 4.3.

We can use the classic tree terminology also for $\mathcal{T}$. We have different hierarchy levels ℓ, parent nodes, children nodes, leaves, etc. In contrast to G, we define $\mathcal{T}$ such that it always has a single root from which we can access all other parts of the temporal tree, e.g., by following the hierarchical edges $\mathcal{E}_N$.

While the tree is temporally evolving, it may undergo structural changes. We call them *topological events* and they are the *appearance* of a node, the *disappearance* of a node, the *merge* of several nodes, and the *split* into several nodes. Merges and splits are the most important topological events for computing the layout, since they require that the involved nodes are right next to each other. A merge and a split event are indicated in Figure 3.

It is straightforward to convert G into $\mathcal{T}$ and vice versa. We convert G into $\mathcal{T}$ by eliminating direct temporal correspondences. Two nodes $\mathbf{u}, \mathbf{v}$ have a direct temporal correspondence, if the edge $(\mathbf{u}, \mathbf{v}) \in E_T$ is the only outgoing temporal edge of $\mathbf{u}$ and the only incoming temporal edge of $\mathbf{v}$. We aggregate a chain of such nodes and edges into a single node with a time series of data values. We convert $\mathcal{T}$ into G by collecting all time steps from all time series in $\mathcal{T}$ and deaggregating all nodes $\mathbf{n}^{\ell}_{t_a,t_b} \in \mathcal{N}$ into a chain with direct temporal correspondences for all time steps overlapping the range $[t_a, t_b]$.

Note that in graph theory, nodes are considered zero-dimensional structures whereas our aggregated nodes are one-dimensional entities. As such, aggregated nodes can intersect each other when being drawn in the plane, which translates to an edge crossing in the planar embedding of two node-edge chains. In the rest of the paper, we will often just speak of "nodes" when referring to "aggregated nodes."

We also note that all algorithms in this paper can be run on the non-aggregated G or the aggregated $\mathcal{T}$, incurring different computational costs. Most notably, the graph layout optimization in Section 4.2.3 has drastically shorter computation times when running on the aggregated $\mathcal{T}$. We evaluate the runtime aspects of our method in Section 5.

4 Temporal Tree Maps

Our goal is to create static visualizations of time-varying trees including topological events. We use the two-dimensional plane for our layout: one dimension represents time, while the other dimension represents the hierarchical nesting similar to a 1D treemap layout (cf. Figure 2c).

The next section characterizes this as a graph drawing problem and reviews the shortcomings of existing approaches. Section 4.2 follows with our own solution to the problem. Section 4.3 presents our adaptation of cushion treemaps to temporal trees.

4.1 Characterization as a Graph Drawing Problem

Our data structure $\mathcal{T}$ is a hierarchically nested graph. If we consider for a moment each hierarchy level independently, then the nodes in each level form a directed graph with topological events as given by the edges in $\mathcal{E}_T$. This is by itself already a graph drawing problem for each hierarchy level.

The challenging part is due to the hierarchical nesting: the goal is to draw all these graphs inside each other, i.e., the graph of hierarchy level ℓ shall be drawn inside the graph of hierarchy level $\ell - 1$. This means that a topological event in a layer $\ell - 1$ has an impact on the graph drawing in layer ℓ. For example, consider the merge of two nodes in layer $\ell - 1$ at time t_i. Each of them has children, which have to be drawn in layer ℓ such that they are next to each other at time t_i to accommodate the merge of the parents. If the children were not next to each other at time t_i, they would need to intersect other nodes in order to be drawn inside their newly merged parent from t_i onwards. In other words, the hierarchical nesting imposes constraints on the graph drawing in each layer.

Lukasczyk et al. [26] presented the first approach to drawing nested graphs by computing the graph layout in each hierarchy level independently and forcing them to nest inside each other in the final drawing stage. This leads to a large amount of intersections in the final image. The graph layout of each hierarchy level is done using the Graphviz library [17]. The optimized graph layout of this library makes sure to avoid intersections as much as possible in each hierarchy level. But since no knowledge of the hierarchical nesting is given, the graph layout cannot accommodate the constraints from other hierarchy levels. The final drawing stage then uses only the graph layout of $\ell = 0$ directly. All other levels force the children to live inside their parents as follows: for a parent $\mathbf{p}$ and a time step t_i, find the children of $\mathbf{p}$ in the graph layout of the children's level and draw them inside $\mathbf{p}$ in the same order as they have been encountered in the children's level. Figure 4 illustrates how the graph layout of each level is done independently and how this causes the intersections in the final drawing stage. This example has been created with the original software of [26].

It should be noted that it is not always possible to draw a graph intersection-free. It is well-known that not every graph can be embedded in the plane [5]. In addition to that, our setting restricts the embedding, namely the nodes are aligned with time steps, i.e., their x-coordinate is fixed. Figure 5 shows an example of this. While an intersection-free layout cannot be expected for every data set, we should strive for a minimal number of intersections. This is the topic of the next section.

4.2 Graph Layout using Constrained Sorting of Leaves

The nested drawing of a temporal tree $\mathcal{T}$ requires the joint consideration of the topological events in each hierarchy level as well as the constraints due to the hierarchical nesting. We bring both requirements into a common setting, where we search for a solution with a minimal number of intersections.

Copyright (c) 2018 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TVCG.2018.2865265

Our approach centers around ordering the leaves of $\mathcal{T}$. We will impose constraints on this order that follow directly from the topological events and the hierarchical nesting of $\mathcal{T}$. We propose a fast heuristic and an optimization approach to find an ordering that fulfills most constraints. Given a leaf order for $\mathcal{T}$, we show how to compute a sorting order for all other nodes of $\mathcal{T}$. Details follow below.

4.2.1 Ordering and Constraints

Let $\mathcal{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ be the set of all leaves of the temporal tree $\mathcal{T}$ with $\mathcal{B} \subseteq \mathcal{N}$. We aim to establish an *ordering* for the leaves, denoted by $\sigma = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, which is essentially an ordered sequence of the elements of $\mathcal{B}$, or a *permutation*. This can be seen as a function where $\sigma(\mathbf{b}_i)$ returns the index of $\mathbf{b}_i$ in the ordering. Since a node of a temporal tree exists only over a certain time span, we introduce the notation $\sigma|_{t_a,t_b}$ as the order of all leaves whose lifetime overlaps with the time span $[t_a, t_b]$. This restricted ordering is directly obtained from the unrestricted ordering σ by removing leaves that do not exist in this time span and keeping the relative order of all other elements. If we want to address the ordering of a subset $\mathcal{G}$ of the leaves, independent of a time span, we use the notation $\sigma|^{\mathcal{G}}$.

We introduce an *ordering constraint* as a tool to require that a certain set of leaves is next to each other in a given time span. More formally, consider a set of leaves $\mathcal{G} = \{\mathbf{g}_1, \ldots, \mathbf{g}_n\}$ which at least partially overlap with the time span $[t_a, t_b]$. The ordering constraint $C = (\mathcal{G}, t_a, t_b)$ is fulfilled if these leaves are next to each other in the corresponding ordering, which can be expressed as

$$|\mathcal{G}| = \max_{\mathcal{G}} \left(\sigma|_{t_a,t_b} \right) - \min_{\mathcal{G}} \left(\sigma|_{t_a,t_b} \right) + 1. \quad (2)$$

Note that we do not require a specific ordering for the leaves in $\mathcal{G}$, only that there is no other leaf between them.

We now have the formalism required to describe the hierarchical nesting of $\mathcal{T}$ and its topological events as ordering constraints. We introduce the following constraints on the leaf order:

- A *hierarchical constraint* imposes that the leaves reachable from an internal node $\mathbf{p}$ are next to each other during the lifetime of $\mathbf{p}$. The internal node $\mathbf{p}$ can be any non-leaf node of $\mathcal{T}$. Considering its lifetime $[t_a, t_b]$, a hierarchical constraint is formally written as $C_H = (\text{leaves}(\mathbf{p}), t_a, t_b)$.
- A *topological constraint* imposes that the leaves reachable from merging and splitting nodes are next to each other at the time step of the event. A merge/split event occurs at time step t_i and includes the nodes $\mathcal{L}$ whose lifetime ends at t_i by merging and/or splitting into the nodes $\mathcal{R}$ whose lifetime starts at t_{i+1}. A topological constraint is formally written as $C_T = (\text{leaves}(\mathcal{L} \cup \mathcal{R}), t_i, t_{i+1})$.

Figure 6 illustrates these constraints using the same example that we used already earlier.

Assume an ordering constraint $C = (\mathcal{G}, t_a, t_b)$ is not fulfilled for a given ordering σ. This means, the leaves $\mathcal{G} = \{\mathbf{g}_1, \ldots, \mathbf{g}_n\}$ are not next to each other because a number of other leaves $\mathcal{H} = \{\mathbf{h}_1, \ldots, \mathbf{h}_m\}$ are mixed in:

$$\sigma|_{t_a,t_b} = (\ldots, \mathbf{g}_1, \ldots, \mathbf{h}_1, \ldots, \mathbf{g}_i, \mathbf{h}_j, \ldots, \mathbf{h}_m, \ldots, \mathbf{g}_n, \ldots). \quad (3)$$

We can always fulfill this constraint, at the possible cost of breaking others, by moving the leaves $\mathcal{H}$ before $\mathbf{g}_1$ and/or after $\mathbf{g}_n$. We identify $m+1$ new sorting orders σ as follows:

$$\begin{aligned} \sigma_0 &= (\ldots, \mathbf{h}_1, \ldots, \mathbf{h}_m, \quad \mathbf{g}_1, \ldots, \mathbf{g}_n, \quad \ldots) \\ &\vdots \\ \sigma_k &= (\ldots, \mathbf{h}_1, \ldots, \mathbf{h}_{m-k}, \quad \mathbf{g}_1, \ldots, \mathbf{g}_n, \quad \mathbf{h}_{m-k+1}, \ldots, \mathbf{h}_m, \ldots) \\ &\vdots \\ \sigma_m &= (\ldots, \quad \mathbf{g}_1, \ldots, \mathbf{g}_n, \quad \mathbf{h}_1, \ldots, \mathbf{h}_m, \ldots). \end{aligned} \quad (4)$$

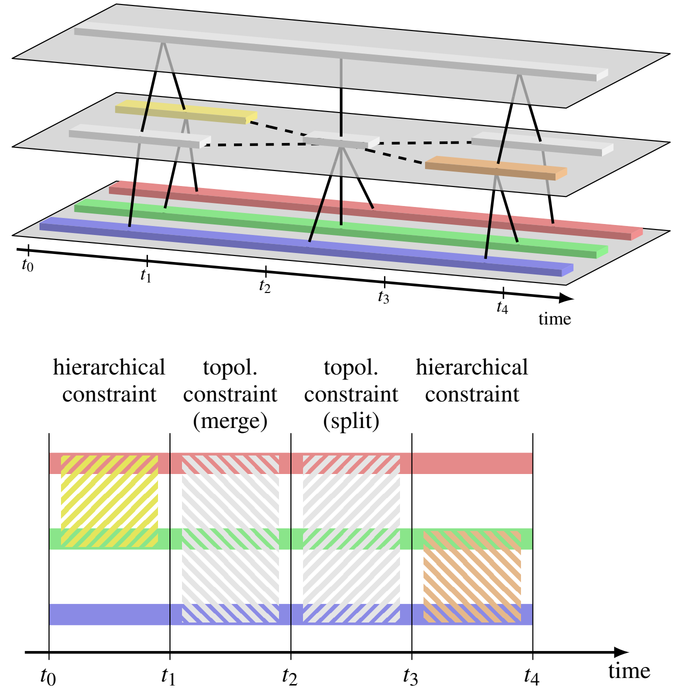


Figure 6. We want to sort the red, green and blue leaves of the temporal tree $\mathcal{T}$. Several constraints are imposed on their ordering due to the hierarchical nesting (yellow and orange constraints) and due to topological events (gray constraints). In order to fulfill a constraint, the involved leaves need to be next to each other in the ordering during the defined time span. Note that we only add a constraint to our system if they are not trivially fulfilled. For example, the root would always impose a hierarchical constraint on all leaves over the entire time, yet any ordering fulfills that. In this example, the two topological constraints are trivially fulfilled as well and would not actually be considered by our method. In fact, just considering the shown two hierarchical constraints suffices to properly sort the leaves. The final nested drawing is shown in Figure 7.

Note how the ordering of the moved leaves $\sigma|^{\mathcal{H}}$ does not change, which may be vital for keeping other constraints intact. Furthermore, the leaves are moved directly in front of $\mathbf{g}_1$ or behind $\mathbf{g}_n$, which also avoids potential conflicts with other constraints as much as possible.

If an ordering σ fulfills all constraints, then the nested drawing of the temporal tree can be done without intersections. This follows directly from the definitions above. If an ordering σ does not fulfill all constraints, then the severity of the visual artifacts depends on the chosen rendering method. We target two rendering methods: the nested drawing akin to Lukasczyk et al. [26] as well as our proposed cushion rendering (Section 4.3). The former will show a certain number of intersections, where an unfulfilled constraint can lead to a single intersection between two curves, or just as well to a larger set of intersections (cf. Figure 4). Our cushion rendering, on the other hand, draws exclusively the leaves as curves without intersection, but unfulfilled constraints show up in the form of visual artifacts in the cushion shading. Figure 5 shows how both rendering approaches deal with violated constraints.

In order to support both rendering methods, we focus on obtaining a σ fulfilling as many constraints as possible. In the following, we propose a heuristic approach that may suffice for simpler examples, and an optimization approach for larger data sets.

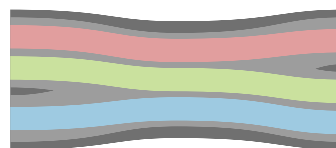


Figure 7. Solving for the hierarchical constraints, the three leaves (See Figure 4) can be properly sorted and will not intersect each other in the nested drawing. Obtained with our heuristic in one single iteration.

Copyright (c) 2018 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TVCG.2018.2865265

4.2.2 Heuristic for Solving Constraints

Consider the leaves of a temporal tree in any given ordering. We record the ordering constraints as discussed in the previous section by iterating over all inner nodes and over all topological events. All unfulfilled constraints are pushed into a *first in, first out* queue (FIFO). We iterate over this queue: after popping the first element from the queue and checking whether it is still unfulfilled, we compute the new sorting orders $\sigma_0, \ldots, \sigma_m$ following (4) as described above. We continue with the sorting order fulfilling most constraints. If we have several of those, we choose one of those randomly. All constraints that became unfulfilled by this procedure are pushed into the queue.[1] We stop this procedure either when the queue is empty or after a certain number of iterations, typically after processing twice as many constraints as we had in the queue to begin with.

In many cases, this heuristic can solve all constraints. Figure 7 shows the computed ordering for the example data set that we have been using throughout this section. The heuristic is able to solve this with one single iteration. In fact, the heuristic is able to find an optimal solution for all examples in this paper except for the data set in Figure 9. Hence, we see this heuristic as a simple-to-implement option for some applications.

In complex data sets such as the one shown in Figure 9 or the large graphs shown in the supplemental material, this heuristic is able to at least drastically reduce the number of unfulfilled constraints. However, it often gets stuck in loops, i.e., it fulfills and breaks the same cycle of constraints over and over again. Nevertheless, it is still useful in these settings for finding a good starting point for the more advanced optimization method introduced in the next section.

4.2.3 Simulated Annealing for Solving Constraints

A leaf order σ is a permutation of the leaves of the temporal tree. If we have n leaves, then we have $n!$ different σ. Our goal is to find a particular leaf order σ violating no or only a small number of ordering constraints. To do so, we apply an optimization method. This requires us to define an *objective function* to assign a measure of quality to each leaf order. With the optimization method we try to find the leaf order minimizing this function.

Let $\{C\}$ be the set of *all* hierarchical and topological ordering constraints. Given a particular leaf order σ, some of them may be violated. We denote the set of *violated* constraints for a given σ with $\{\bar{C}^\sigma\}$. We define the *violation ratio* $v(\sigma)$ as a means to assess the amount of violated constraints in a normalized manner:

$$v(\sigma) = \frac{|\{\bar{C}^\sigma\}|}{|\{C\}|} \qquad 0 \le v \le 1 \; . \tag{5}$$

Note that lower values of v are considered to be better, i.e., we want to minimize that function. The optimization problem can now be stated as

$$\arg\min_{\sigma} \; v(\sigma) \; . \tag{6}$$

We approach it using *Simulated Annealing* [24], which works in a nutshell like this: we start with an initial leaf order σ_0. To compute a new leaf order σ_i in each iteration, we take the one from the previous iteration σ_{i-1} and resolve one random, currently violated constraint. This gives us, according to (4), possibly several new leaf orders of which we randomly choose one, denoted by σ'. If $v(\sigma') \le v(\sigma_{i-1})$, then we set $\sigma_i = \sigma'$ and move on to the next iteration. If, however, $v(\sigma') > v(\sigma_{i-1})$, then the main feature of Simulated Annealing comes into effect: the new leaf order is worse than the previous one, but it *may* be accepted by Simulated Annealing in an attempt to not get stuck in local minima or loops. This is steered by a parameter called *temperature* T, which is initialized with a high value and then slowly decays with a factor $0 < d < 1$ to be applied every k iteration steps:

$$T_0 = T_{\text{init}}, \; T_i = d\, T_{i-1} \; . \tag{7}$$

[1] Some constraints may have become fulfilled by this procedure as well. They are still in the queue and will be removed when they appear at the front of the queue.

In this setting, we are more likely to choose a worse new leaf order if the temperature is high. The probability for accepting a worse leaf order is computed as

$$p = e^{\frac{v(\sigma_{i-1}) - v(\sigma')}{T}} \; . \tag{8}$$

The process stops once the temperature drops below a near-zero threshold, or a maximal number of iterations is reached, or all constraints are fulfilled.

We initialize this optimization with a leaf order obtained with the heuristic from the previous section. Simulated annealing has always been able to improve on that initial ordering. The running times range from a few seconds to minutes, depending on the size of the problem and the initial temperature. We evaluate this approach in Section 5 in detail.

4.2.4 Sorting Order for all Nodes

Drawing a temporal tree requires an ordering of all nodes, which can be computed from the leaf order σ straightforwardly: for each inner node $\mathbf{p}$ of $\mathcal{T}$, find the first leaf in σ reachable from $\mathbf{p}$ and use this index to sort $\mathbf{p}$ among the other nodes from the same hierarchy level.

4.3 Adaptation of Cushion Maps

In many application cases (e.g., filesystem), the data value at each parent node is the sum of the data of its children, see Equation (1). If the drawing is supposed to represent the data truthfully, then the entire space of a parent is consumed by its children. In fact, the leaves of the entire tree consume the entire drawing space, since their data sums up to the data value of the root. A nested drawing with space for the parents as in Figure 7 is not the best choice for such application cases.

We propose a drawing scheme for temporal trees taking into account exactly those application cases. It is an adaptation of the classic cushion treemaps [38], which computed the value of the cushion for each pixel on the CPU. We want to exploit graphics hardware and utilize the GPU for this. Hence, we will triangulate the leaves of the tree and provide each vertex with enough information such that the cushions can be computed on the GPU in a shader. We will detail this in the following.

First, we need to identify all time steps in which any data value changes in the temporal tree. This can easily be obtained by iterating over all leaves and collecting the time steps of their time series. This globally unique and sorted list of time steps is used to divide the time axis (x-axis) of the drawing. Furthermore, we obtain the sum of all leaves' data values for these time steps. This can be used to normalize the values in each time step.

As just observed, it suffices to draw the leaves as they sum up to the data value of the root. Given the optimized leaf order σ, we start with the first leaf $\mathbf{b}_1$. It extends over the time span $[t_a, t_b]$. We place a triangle strip at the bottom of the drawing space. The x-coordinates are given by the global time steps in $[t_a, t_b]$. The y-coordinates are given by the normalized data value in each time step. We add a triangle strip for every subsequent leaf in the same manner, except that they attach to the top of previously drawn leaves.

Topological events need special treatment. Our goal is to indicate the merge or split by means of a merging/splitting highlight curve in the final cushion rendering. We make the transition as follows: consider a split event where a leaf $\mathbf{b}_1$ splits into leaves $\mathbf{b}_2, \mathbf{b}_3$. Due to our data structure, $\mathbf{b}_1$ ends directly at the time of the split, while $\mathbf{b}_2, \mathbf{b}_3$ start there. Also, they match in size, i.e., $d(\mathbf{b}_1) = d(\mathbf{b}_2) + d(\mathbf{b}_3)$. We insert a new vertex a bit before the event into the triangle strip of $\mathbf{b}_1$. It is placed directly on top of the cushion highlight and connected to the regular first vertices of $\mathbf{b}_2, \mathbf{b}_3$. This way, we impose a split on the cushion highlight following the split in the data. This works just as well for merges and also for events with more splits/merges than two.

Once the triangulation is done, we provide cushion information to each vertex. Cushion rendering [38] has the goal to communicate hierarchical information even in situations where the parents cannot be seen. The main idea is to simulate a virtual "landscape" that reveals the nesting information through shading. Basically, each node of the tree is assigned a parabola with the height depending on the hierarchy level

Copyright (c) 2018 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TVCG.2018.2865265

of the node. Assume a node is placed between $[y_0, y_1]$ in a given time step, then the parabola is given as:

$$\Delta z(y) = \frac{4 f^{\ell} h \, (y - y_0)\,(y_1 - y)}{y_1 - y_0} \,, \tag{9}$$

where h is a base height for all cushions and $0 < f < 1$ determines the height decay over increasing hierarchical levels. The final cushion "landscape" is the sum of the parabolas of all nodes. Thankfully, the sum of two parabolas is a parabola itself! Hence, we can describe the final cushion "landscape" by assigning an appropriately summed-up parabola to each leaf. Technically, we store the parabola parameters at the vertices of the leaf triangulation and use this to perform shading calculations on the GPU.

Lastly, we also use color to indicate the hierarchical nesting. To do so, we traverse $\mathcal{T}$ starting at the root, visiting each node and distributing colors between user-defined hierarchy levels. The colors are chosen in different ways, e.g., randomly, along the hue axis of the HSV color model, or using different colorbrewer schemes [7].

4.4 Interacting with Temporal Treemaps

Our final visualization comes with a number of interactive elements. Aspects such as color, cushions, lighting conditions, node selection can be modified interactively after the finalization of the order.

Through exploiting GPU shading procedures, we can interactively change light attributes such as its direction or ambient/diffuse colors. For the cushions, varying the parameters h and f highlights different aspects of the hierarchy. A lower factor f puts more focus on the first hierarchy layers, whereas higher values emphasize deeper parts. Changes in height h lead to overall flatter or steeper height profiles. We can also choose a depth range for coloring and cushioning. This allows to explore the data through focusing on selected hierarchical layers. Further filtering on the time span and toggling of first level hierarchy nodes is supported as well.

We refer the reader to the supplemental video where these interactions are showcased.

5 Evaluation

5.1 Comparison to Lukasczyk et al. [26]

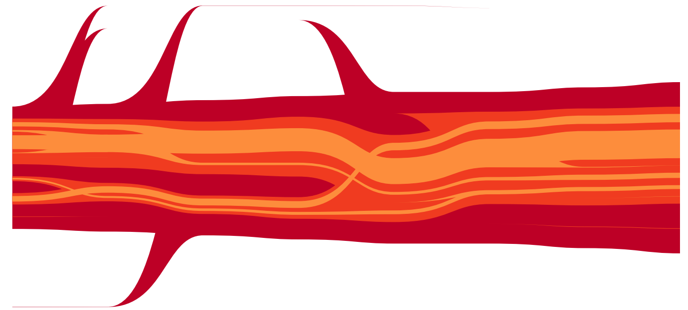
We start with a comparison using a closeup of the *Viscous Fingers* data set, which was used in the publication of the original method for Nested Tracking Graphs [26]. We thank the authors for making their code and data publicly available. Figure 8 shows a side-by-side comparison of the graph layouts. It can easily be seen that our method creates an intersection-free layout, while the original method fails to do so.

Figure 9 tells a similar story, except that this data has more time steps and shows even more intersections for the original method. The underlying time-dependent data, which we will refer to as the *Cylinder* data set, describes vortex activity in the wake of a square cylinder by means of the Okubo-Weiss criterion [12, 39]. We look at a part of a temporal zoom-in of 15 time steps.

In Table 1 we compare the data structures employed by either method in a quantitative manner. The compression ratio of our new aggregated temporal tree $\mathcal{T}$ is highest for data sets where only a few nodes change per time step. The *Pyhton* data set is an example for this as it records all changes to the Python source code file tree by scanning over 100k commits to the repository over a time range from the 1990s to now.

The most important advantage of the aggregation are the drastically reduced computation times of our algorithm. For example, our method needs 30 seconds for the aggregated version of the *Cylinder* data set with its total of 508 time steps. The non-aggregated version incurs a running time of 30 minutes.

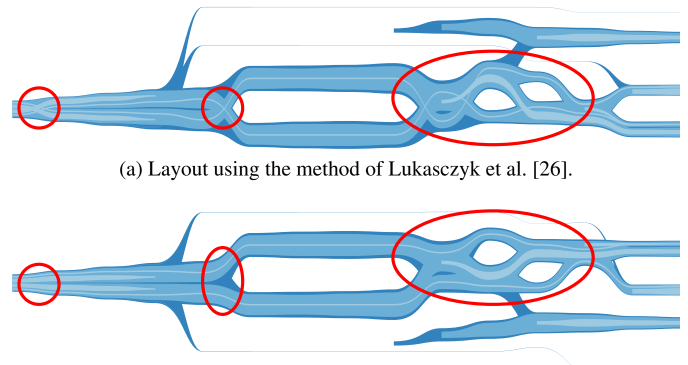
We deem it unfeasible and not constructive to compare our method and the method of Lukasczyk et al. [26] in terms of computation times. We employ drastically different technologies. While Lukasczyk et al. [26] call Graphviz for most of the layout, the final drawing happens in Javascript in the browser. In our case, everything is done native in C++. Still, in neither case, a user has to wait longer than half a minute to get a result.



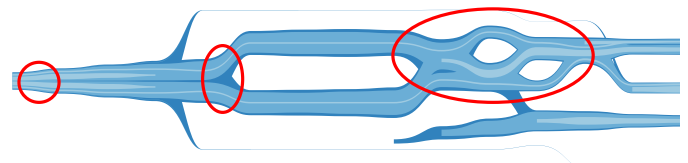
(a) Layout using the method of Lukasczyk et al. [26].

(b) Layout using our algorithm.

Figure 8. Closeup of the *Viscous Fingers* data set from [26]. We zoomed into the time range $[40, 47]$ to highlight the intricate structures. Note how our algorithm produces an intersection-free layout, whereas some bands are intersecting each other for the original method.



(a) Layout using the method of Lukasczyk et al. [26].



(b) Layout using our algorithm.

Figure 9. Layout of a nested tracking graph in the *Cylinder* data set. The red circles indicate regions where the original method for Nested Tracking Graphs [26] produces a layout with many intersections. Our method is able to produce a layout with just one intersection. We are looking at a temporal and spatial zoom-in covering about 15 time steps. A larger, zoomed-out version is shown in the supplemental material.

Nevertheless, it should be noted that our method needed less than a second for the results shown in Figures 8b and 9b.

Data Set	Aggregated Data Structure $\mathcal{T}$				Non-Aggregated Data Structure G			
	$\lvert\mathcal{N}\rvert$	$\lvert\mathcal{E}_N\rvert$	$\lvert\mathcal{E}_T\rvert$	Memory	$\lvert N\rvert$	$\lvert E_N\rvert$	$\lvert E_T\rvert$	Memory
Viscous Fingers	379	589	344	0.01 MB	919	918	884	0.03 MB
Cylinder	7094	10236	3923	0.24 MB	28904	31539	21810	0.92 MB
Python	11643	11642	0	0.22 MB	372127	372025	360484	12.6 MB

Table 1. The aggregated data structure introduced in Section 3 has a significantly lower footprint, especially for filesystem data sets such as the *Python* data set.

Copyright (c) 2018 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TVCG.2018.2865265

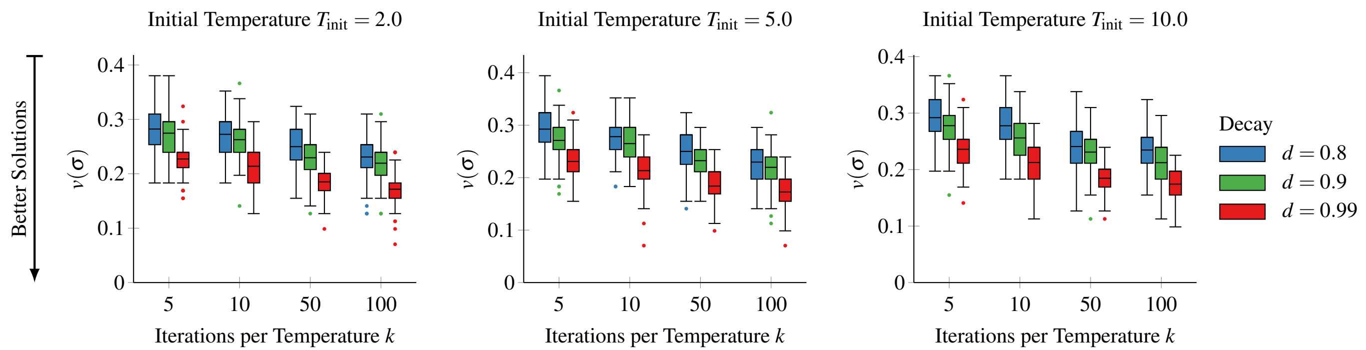


Figure 10. This parameter study for the Simulated Annealing method shows that, on average, it behaves quite stable. We investigated three initial temperatures T_0 (2, 5, 10), four settings for the number of iterations k until a temperatures decay occurs (5, 10, 50, 100), and three distinct settings for the temperature decay factor d (0.8, 0.85, 0.9). See Equation (7). Each bar summarizes 100 runs for the respective setting. The plots show that very good results can be obtained with any of the presented settings, but the spread of the individual runs should not be neglected. We propose to run several times, possibly in parallel, since a single run takes less than a second.

5.2 Evaluation of Simulated Annealing

Figure 11 plots the number of violated constraints during a run of the Simulated Annealing method on the *Cylinder* data set shown in Figure 9b. The method starts with only a few violated constraints, but after about 25 iterations and with still high temperature it chooses to worsen the condition and break some constraints in an attempt to find a globally better solution. This is a typical behavior of Simulated Annealing and actually one of its advantages. Once the temperature cools down, the fluctuations lessen and the method found a good solution around iteration 350. This whole procedure takes less than a second.

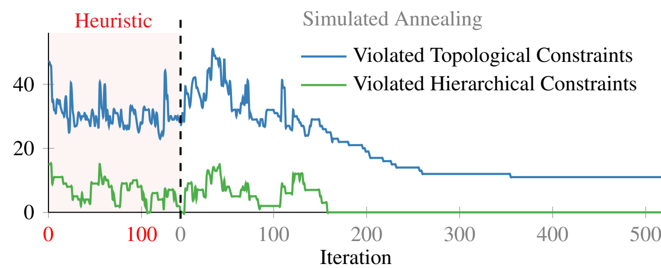
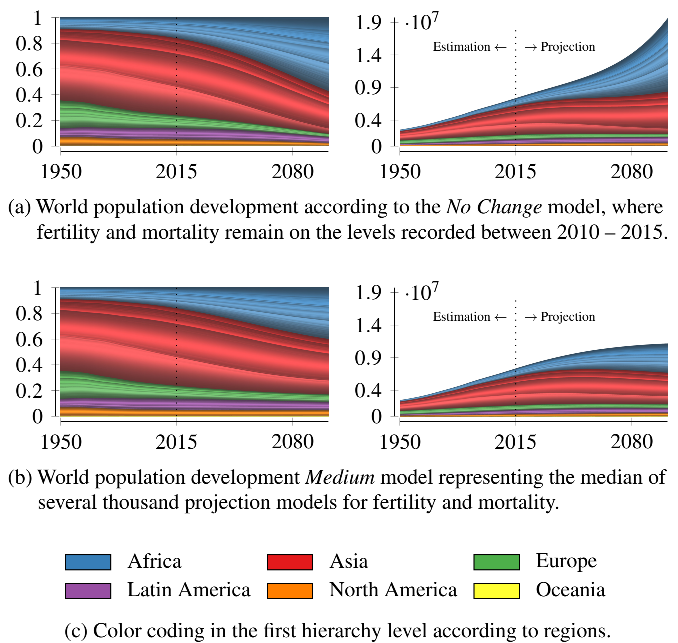


Figure 11. We fist run the heuristic for twice as many iterations as there are constraints. The approach then may choose to run into worse conditions (violate more constraints) during the Simulated Annealing part in order to not get stuck in local minima. Still, it found a configuration after 355 iterations that violates only a few topological constraints. This is a run on the *Cylinder* data set shown in Figure 9b.

We conducted a parameter study for the very same data set. We investigated three initial temperatures, four settings for the number of iterations until a temperature decay occurs, and three settings for the temperature decay factor. We ran Simulated Annealing for the entire so-defined 3D parameter space and Figure 10 shows the three two-dimensional projections of the results. The plots show that very good results can be obtained with many different settings. We observe a slight advantage of parameter settings with longer run times, i.e. higher initial temperature, more iterations per temperature and higher decay. On average, the method behaves nicely and leads to stable results. However, since it uses randomness by design, the spread of the individual runs needs to be accounted for. For example, it is advisable to run the method several times and choose the best result. This can be done in parallel. Our implementation is currently not parallel and requires less than a second on current hardware.



(a) World population development according to the *No Change* model, where fertility and mortality remain on the levels recorded between 2010 – 2015.

(b) World population development *Medium* model representing the median of several thousand projection models for fertility and mortality.

(c) Color coding in the first hierarchy level according to regions.

Figure 12. Different types of data normalization can be used with our cushion rendering method. The data in the left figures is normalized per time step, whereas the right figures show the data in absolute terms. The plots show the development of the world population until the year 2100 according to a United Nations report [35].

6 Results

Besides the already shown results, we would like to showcase the utility of our approach with three more data sets.

Different types of data normalization can be used with our method. Figure 12 exemplifies this showing the development of the world population until the year 2100 as projected by the United Nations in *World Population Prospects: The 2017 Revision* [35] according to different models making assumptions on fertility, mortality and net migration. We show two of the nine variants here; see the supplemental material for all variants and details. Figure 12a shows the population development assuming that fertility and mortality remain unchanged. The left image shows the data with a normalization per time step. This utilizes the entire visualization space and supports relative size comparisons: Europe's fraction of the world population declines over the entire time span, Asia's fraction declines since the early 21st century, whereas

Copyright (c) 2018 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TVCG.2018.2865265

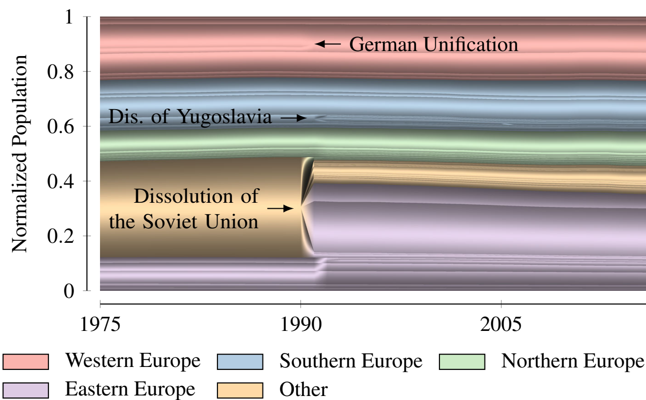


Figure 13. Development of Europe's population between 1975 and 2016. A number of splitting and merging events on the country level occur as highlighted by the cushion transitions.

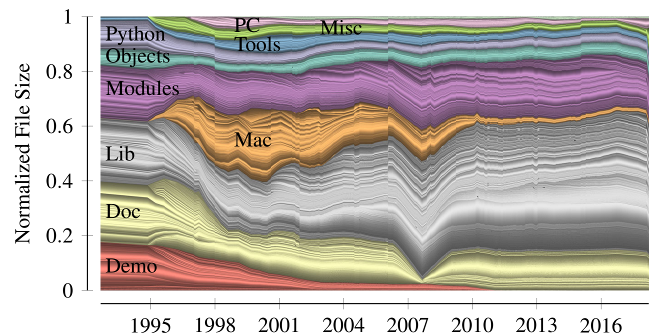


Figure 14. Evolution of the Python repository from 1992 until now. We can see the major development trends with some folders becoming very large and supposedly important over time.

more than every second human will live in Africa by the year 2100. The right image in Figure 12a shows the same data in absolute terms (normalized to the global maximum), which reveals that the absolute population numbers are constant for all regions at least since the early 21st century, except for Africa experiencing a strong increase. Note that this model is considered to be unrealistic. The most likely model according to [35] is shown in Figure 12b. The right image reveals that the world population levels out at around 11 billion people.

The data in the previous example does not contain topological events, even though an accurate depiction of history would show splitting and merging countries. Especially Europe has been subject to a number of such cases in the 20th century. We use the latest available demographic data [34] for Europe's population between 1975 and 2016 and adhere to historical events such as the German unification in 1990 and the dissolution of the Soviet Union in 1991. The result in Figure 13 shows how the cushion rendering is effective in conveying these events.

Figure 14 shows the development of the *CPython* source code repository [13] since its early development. The covered time span is between August 1992 and March 2018 obtained by sampling every 1000th of the 101181 commits. The colored cushion rendering is effective in revealing two interesting events. First, the relative size of the `Doc` folder first decreases and then increases tremendously during 2007: the Python documentation switched from LaTeX to *reStructuredText* and during a brief period the `Doc` folder does not exist at all. Second, the `Mac` folder seems to be added only around 1995. Upon further investigation, this folder exists from the very beginning, albeit almost empty. An introduction on how to use Python on Mac is introduced in August 1994. Figure 14 shows that our rendering method can be used to identify major trends in such data. A shortcoming of our current implementation can be seen as well. We do not apply any form of smoothing, which makes some parts of the plot rather discontinuous. Furthermore, it is yet to be determined, e.g., through a user study, whether the cushions alone allow for perception of the hierarchical nature of the data. We suspect the combination of color and cushions is needed.

7 Conclusions and Future Work

We presented a novel layout algorithm for temporally evolving trees with changing topology and data. Based on a combination of a heuristic and simulated annealing, it produces a layout with as few intersections as possible. Our evaluation has shown that the algorithm runs fast, which is also fostered by our new data structure for temporally evolving trees, recording only the changes to the tree. Our new cushion-based rendering scheme highlights temporal evolution and hierarchical nesting at the same time.

Our current objective function in Equation (5) essentially only counts the number of violated constraints. It is an interesting avenue for future research to incorporate more aspects such as the severity of the visual artifact of an unfulfilled constraint depending on the chosen rendering method, the associated data values (thickness in the final drawing), or the wigglyness of the final drawing similar to how Bartolomeo and Hu [2] did this for stream graphs.

We excluded cases where a child changes its parent. This happens often in filesystems: a file is moved from one folder to another. In Figure 13 we have modeled this case for the Soviet Union by splitting the parent into two separate nodes: one before the move and one after the move. However, for a data set with many moving nodes, the introduction of splits diminishes some of the efficiency gained by our data structure. A direct support for moving a node to a different parent is desirable, yet any such move leads to an intersection in the drawing, since the child needs to cross over to the other parent. Nonetheless, one may be able to minimize the number of intersected bands.

Furthermore, we believe the navigation of our visualization can be enhanced by connecting it to a snapshot visualization through linking and brushing. Lukasczyk et al. [26] display the isosurfaces for a hierarchy level alongside the Nested Tracking Graph allowing for selection and highlighting of components. For the other types of data sets (file systems, world population), a 2D treemap or a world map can serve to visualize the state of the chosen time step in a different context.

Acknowledgments

This work was supported through grants from the Swedish Foundation for Strategic Research (SSF) and the Swedish e-Science Research Centre (SeRC). The presented concepts have been implemented in the Inviwo framework.

References

[1] M. Balzer and O. Deussen. Voronoi treemaps. In *Proceedings IEEE Symposium on Information Visualization*, pages 49–56, Washington, DC, USA, 2005. IEEE Computer Society. 2
[2] M. D. Bartolomeo and Y. Hu. There is more to streamgraphs than movies: Better aesthetics via ordering and lassoing. *Computer Graphics Forum*, 35(3):341–350, 2016. 3, 9
[3] D. Baur, B. Lee, and S. Carpendale. Touchwave: Kinetic multi-touch manipulation for hierarchical stacked graphs. In *Proceedings ACM International Conference on Interactive Tabletops and Surfaces*, pages 255–264, New York, 2012. ACM. 1, 3
[4] F. Beck, M. Burch, S. Diehl, and D. Weiskopf. A taxonomy and survey of dynamic graph visualization. *Computer Graphics Forum*, 36(1):133–159, Jan. 2017. 3
[5] J. A. Bondy and U. S. R. Murty. *Graph theory with applications*. Elsevier Science Publishing Co., Inc., New York, 1976. 4
[6] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976. 3
[7] Brewer, Cynthia A., 2018. http://www.ColorBrewer.org, accessed 2018. 7
[8] M. Bruls, K. Huizing, and J. J. van Wijk. Squarified treemaps. In *Proceedings Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42. Eurographics Association, 2000. 2

Copyright (c) 2018 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TVCG.2018.2865265

[9] K. Buchin, M. J. van Kreveld, H. Meijer, B. Speckmann, and K. Verbeek. On planar supports for hypergraphs. *Journal of Graph Algorithms and Applications*, 15(4):533–549, 2011. 3
[10] M. Burch, C. Vehlow, F. Beck, S. Diehl, and D. Weiskopf. Parallel edge splatting for scalable dynamic graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2344–2353, 2011. 3
[11] L. Byron and M. Wattenberg. Stacked graphs - geometry & aesthetics. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1245–1252, 2008. 3
[12] S. Camarri, M.-V. Salvetti, M. Buffoni, and A. Iollo. Simulation of the three-dimensional flow around a square cylinder between parallel walls at moderate reynolds numbers. In *XVII Congresso di Meccanica Teorica ed Applicata*, 2005. 7
[13] CPython repository, `https://github.com/python/cpython`. 9
[14] E. Cuenca, A. Sallaberry, F. Y. Wang, and P. Poncelet. Multistream: A multiresolution streamgraph approach to explore hierarchical time series. *IEEE Transactions on Visualization and Computer Graphics*, 2018. 1, 3
[15] W. Cui, S. Liu, Z. Wu, and H. Wei. How hierarchical topics evolve in large text corpora. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2281—2290, December 2014. 3
[16] M. de Berg, K. Onak, and A. Sidiropoulos. Fat polygonal partitions with applications to visualization and embeddings. *Journal of Computational Geometry*, 4(1):212–239, 2013. 2
[17] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, 2000. 3, 4
[18] J. Görtler, C. Schulz, D. Weiskopf, and O. Deussen. Bubble treemaps for uncertainty visualization. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):719 –728, Jan. 2018. 2
[19] S. Hahn, J. Trümper, D. Moritz, and J. Döllner. Visualization of varying hierarchies by stable layout of voronoi treemaps. In R. S. Laramee, A. Kerren, and J. Braz, editors, *Proceedings of the 5th International Conference on Information Visualization Theory and Applications*, pages 50–58. SciTePress, 2014. 1, 3
[20] M. T. Hajiaghayi and Y. Ganjali. A note on the consecutive ones submatrix problem. *Information Processing Letters*, 83(3):163–166, 2002. 3
[21] S. Havre, E. G. Hetzler, P. Whitney, and L. T. Nowell. Themeriver: Visualizing thematic changes in large document collections. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):9–20, 2002. 3
[22] W.-L. Hsu. A simple test for the consecutive ones property. In *Proceedings of the Third International Symposium on Algorithms and Computation*, pages 459–468, London, UK, UK, 1992. Springer-Verlag. 3
[23] B. Johnson and B. Shneiderman. Tree maps: A space-filling approach to the visualization of hierarchical information structures. In G. M. Nielson and L. J. Rosenblum, editors, *Proceedings IEEE Visualization*, pages 284–291. IEEE Computer Society Press, 1991. 2
[24] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. 6
[25] S. Liu, Y. Wu, E. Wei, M. Liu, and Y. Liu. Storyflow: Tracking the evolution of stories. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2436–2445, Dec 2013. 3
[26] J. Lukasczyk, G. Weber, R. Maciejewski, C. Garth, and H. Leitte. Nested tracking graphs. *Computer Graphics Forum (Proceedings EuroVis)*, 36(3):643–667, 2017. 2, 3, 4, 5, 7, 9
[27] P. Neumann, S. Schlechtweg, and M. S. T. Carpendale. Arctrees: Visualizing relations in hierarchical data. In *Proceedings Joint Eurographics - IEEE VGTC Symposium on Visualization*, pages 53–60. Eurographics Association, 2005. 2
[28] B. Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on Graphics*, 11(1):92–99, Jan. 1992. 1, 2
[29] B. Shneiderman and M. Wattenberg. Ordered treemap layouts. In K. Andrews, S. F. Roth, and P. C. Wong, editors, *Proceedings IEEE Symposium on Information Visualization*, pages 73–78. IEEE Computer Society, 2001. 2
[30] M. Sondag, B. Speckmann, and K. Verbeek. Stable treemaps via local moves. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):729–738, 2018. 1, 3
[31] A. Sud, D. Fisher, and H. Lee. Fast dynamic voronoi treemaps. In M. A. Mostafavi, editor, *Proceedings Seventh International Symposium on Voronoi Diagrams in Science and Engineering*, pages 85–94. IEEE Computer Society, 2010. 1, 3
[32] S. Tak and A. Cockburn. Enhanced spatial stability with hilbert and moore treemaps. *IEEE Transactions on Visualization and Computer Graphics*, 19(1):141–148, 2013. 1, 3
[33] Y. Tu and H. Shen. Visualizing changes of hierarchical data using treemaps. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1286–1293, 2007. 1, 3
[34] United Nations. *United Nations Demographic Yearbooks 1984–2016*. United Nations, New York, 1984–2018. 9
[35] United Nations. World population prospects: The 2017 revision, 2017. 8, 9
[36] T. C. van Dijk, M. Fink, N. Fischer, F. Lipp, P. Markfelder, A. Ravsky, S. Suri, and A. Wolff. Block crossings in storyline visualizations. In Y. Hu and M. Nöllenburg, editors, *Graph Drawing and Network Visualization*, pages 382–398, Cham, 2016. Springer International Publishing. 3
[37] R. van Hees and J. Hage. Stable and predictable voronoi treemaps for software quality monitoring. *Information and Software Technology*, 87:242–258, July 2017. 1, 3
[38] J. J. van Wijk and H. van de Wetering. Cushion treemaps: Visualization of hierarchical information. In *Proceedings IEEE Symposium on Information Visualization*, pages 73–78. IEEE Computer Society, 1999. 2, 6
[39] W. von Funck, T. Weinkauf, H. Theisel, and H.-P. Seidel. Smoke surfaces: An interactive flow visualization technique inspired by real-world flow experiments. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization 2008)*, 14(6):1396–1403, November - December 2008. 7
[40] M. Wattenberg and J. Kriss. Designing for social data analysis. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):549–557, July - August 2006. 1, 3

Copyright (c) 2018 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.