



Improving Temporal Treemaps by Minimizing Crossings

Alexander Dobler¹  and Martin Nöllenburg¹ ¹TU Wien, Algorithms and Complexity Group, Austria

Abstract

Temporal trees are trees that evolve over a discrete set of time steps. Each time step is associated with a node-weighted rooted tree and consecutive trees change by adding new nodes, removing nodes, splitting nodes, merging nodes, and changing node weights. Recently, two-dimensional visualizations of temporal trees called temporal treemaps have been proposed, representing the temporal dimension on the x -axis, and visualizing the tree modifications over time as temporal edges of varying thickness. The tree hierarchy at each time step is depicted as a vertical, one-dimensional nesting relationships, similarly to standard, non-temporal treemaps. Naturally, temporal edges can cross in the visualization, decreasing readability. Heuristics were proposed to minimize such crossings in the literature, but a formal characterization and minimization of crossings in temporal treemaps was left open. In this paper, we propose two variants of defining crossings in temporal treemaps that can be combinatorially characterized. For each variant, we propose an exact optimization algorithm based on integer linear programming and heuristics based on graph drawing techniques. In an extensive experimental evaluation, we show that on the one hand the exact algorithms reduce the number of crossings by a factor of 20 on average compared to the previous algorithms. On the other hand, our new heuristics are faster by a factor of more than 100 and still reduce the number of crossings by a factor of almost three.

Keywords: Temporal treemaps, crossing reduction, temporal data, algorithm engineering, computational experiments

CCS Concepts

• Human-centered computing → Treemaps; Graph drawings; • Theory of computation → Design and analysis of algorithms;

1. Introduction

Trees are a very common and useful way to represent hierarchical data such as file systems, source code repositories with versioning control, structures of organizations, biological taxonomies etc. By adding data values as node weights in the tree, we can also represent numeric attributes of the underlying data. There is a plethora of tree visualization techniques [Sch11], with treemaps [Shn92] being one of the most well-known ones for weighted trees.

In many contexts such trees may change over time, leading to *temporal trees*. First, the data values associated with the nodes may change, e.g., by changing the contents of a file or the size of a research group in a university. Second, the tree structure itself may change over time, e.g., by creating, deleting, or moving files, by splitting or merging research units or by restructuring institutes in the faculty of a university, and so on. This leads to a temporal sequence of related trees, one for each discrete time step. The evolution of the tree between two time steps is modelled by *temporal edges* that represent the correspondence between two nodes of consecutive trees. A node might have multiple outgoing and incoming temporal edges, representing the splitting or merging of a node, respectively. Naturally, each node in a tree has a level representing its depth in this tree. Refer to Figure 1a for an example. Visualizing

temporal trees is a challenging task as we have to convey the tree structure for each point in time, and also represent the changes that occur over time. Many visualization techniques for temporal trees have been proposed [BBDW17, GLAK23].

This paper deals with *temporal treemaps*, a temporal tree visualization approach [KW19] that is based on the *nested tracking graph* metaphor [LWM*17]. The x -axis of a temporal treemap represents time, temporal edges are drawn as horizontal x -monotone bands of varying thickness depending on node weights, and the hierarchical relationships are conveyed by vertical *nestings* of these temporal edges (see Figure 1c). The only restriction imposed on temporal trees by this temporal treemap visualization is that temporal edges can only connect nodes of the same level, i.e., nodes cannot change their level over time. The similarity to treemaps – and thus the origin of the name *temporal treemaps* – is immediate when considering the tree of Figure 1a in the first time step t_1 visualized as a vertical treemap in Figure 1b. One could interpret Figure 1b as looking at the temporal treemap in Figure 1c from the left side.

The bands in temporal treemaps can cross in the visualization, and it is non-trivial how to avoid such crossings. For example, the visualization in Figure 1c has a crossing between the two temporal edges (a_1, b_2) and (a_2, b_1) . Such crossings depend on the vertical

order of temporal edges, which is not pre-specified in the input data and hence must be computed. One of the contributions of Köpp and Weinkauff [KW19] are heuristic algorithms to compute such orders that lead to few crossings. However, they do not provide a formal way of quantifying the number of crossings in the output of their algorithms, nor do they investigate their exact minimization.

We tackle the problem of minimizing crossings in temporal treemaps for the first time from a rigorous algorithm engineering perspective with the aim of improving the existing heuristic approaches of Köpp and Weinkauff [KW19] for temporal treemap visualizations. We note that such visualizations have already been motivated extensively by Köpp and Weinkauff and Lukaszczuk et al. [LWM*17], and their relevance in visualization is witnessed by the multitude of follow-up works. This paper aims to improve temporal treemap (and nested tracking graph) visualizations by providing algorithms that minimize crossings more effectively and more efficiently. In fact, crossings are one of the most important factor that negatively influences graph readability [Pur97, WPCM02] and hence crossing minimization is a well established way of improving graph visualizations and reducing visual clutter. Our main contributions include the following.

- In Section 4 we formally define two types of crossings occurring in temporal treemaps that rely on a combinatorial characterization of such visualizations. We define two associated computational problems of minimizing the two types of crossings.
- In Section 5.1 we show that both problems are generally NP-hard. In the supplementary material [DN23] we give polynomial algorithms that determine if a solution with zero crossings exists.
- In Section 5.2 we discuss the algorithms by Köpp and Weinkauff [KW19] in more detail. We observe some drawbacks in those algorithms that we want to circumvent in our algorithms.
- In Section 5.3 we give exact algorithms that compute optimal solutions with the minimum number of crossings.
- In Section 5.4 we give a set of new heuristic algorithms.
- In Section 6 we conduct an experimental evaluation of the existing and new algorithms.
- Finally, in Section 7, we showcase the effect of more rigorous crossing minimization in an example for a real-world dataset.

The implementations of algorithms in this paper make use of the code-base of Köpp and Weinkauff [KW19]. All implementations, datasets, additional experimental evaluations, and additional examples of temporal treemaps are available on OSF [DN23].

2. Related work

In this section we discuss related work starting with brief overviews on tree and temporal data visualizations focusing on the most relevant aspects such as treemaps and temporal streams. We then discuss in more detail approaches on temporal tree visualization as well as work on crossing minimization in layered graphs.

Tree visualizations. For an overview of the rich literature on tree visualization we refer to the treevis.net bibliography [Sch11]. Tree data with nodes having real-valued weights such that the weights of internal nodes are at least the sum of weights of their children, appear in many contexts such as file systems with files, folders, and

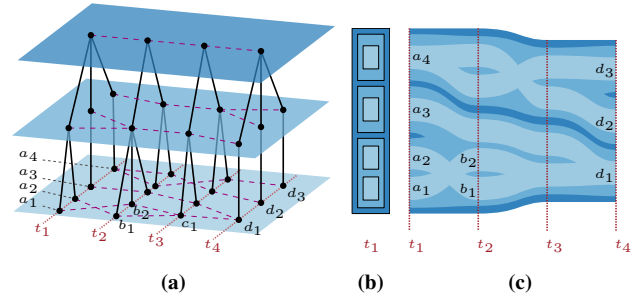


Figure 1: (a) A temporal tree \mathcal{G} over four time steps. Tree edges are drawn in solid black, temporal edges in dashed violet. Layers are displayed as stacked blue planes, with black nodes placed on them. Lower levels have lighter color. Leaves of the first and last time steps are labelled. (b) The tree at time t_1 drawn as a vertical treemap. (c) \mathcal{G} drawn as a temporal treemap with two crossings. Temporal edges in lower levels have lighter color. The leaves of the first and last time steps are again labelled. The combinatorial layout corresponds to the vertical ordering of the leaves at t_1 to t_4 .

their respective sizes on the hard disk. Such data are often visualized as *treemaps* [JS91, Shn92] in a nested, space-filling 2d layout. A node corresponds to a (rectangular) region of the visualization and its weight is represented by the size of the region. Hierarchical relationships are conveyed by nesting of those regions. Many variants of treemaps have been proposed, with neighborhood constraints [SW01], varying region shapes [BHvW00, BD05, dBOS13, GSWD18] and types of region boundaries [vVvdW99].

Temporal data visualizations. Temporal data are usually represented by a sequence of data values for a discrete set of time steps. For example, a set of *subjects* (e.g. countries) might have a data value (e.g. population) for each time step (e.g. year). For a general overview of visualizing temporal data we refer to the book by Aigner et al. [AMST23]. Such data might be visualized in a static way by letting the x -axis represent time and the varying data values are represented by stacked horizontal bands with varying widths for each subject. Such visualizations are known as *streamgraphs* or *stacked graphs*. Visualization systems for these are available. One of the first is ThemeRiver [HHWN02]; other works focus on optimizing aesthetic criteria such as wiggles [BH16, BW08].

Temporal tree visualizations. Temporal trees are trees that change over a sequence of discrete time steps. Many visualizations for such data exist, e.g., stable sequences of node link diagrams for trees evolving by node insertion and deletion [GLAK23]. Adaptations of treemap layout algorithms aim at producing similar treemap visualizations for similar trees and use animation to show the changes over time: Hahn et al. [HTMD14], Sud et al. [SFL10], Hees and Hage [vHH17] adapt Voronoi treemap algorithms, while Sondag et al. [SSV18], and Tak and Cockburn [TC13] adapt algorithms for rectangular treemaps. Vernier et al. [VSC*20] provide a quantitative evaluation of these time-dependent treemap visualizations. Adaptations of streamgraphs attempt to visualize such data by conveying hierarchy with color [CSWP18, WK06], or showing individual hierarchy levels separately [BLC12]. Burch

et al. [BBLW14] propose a visualization of changing hierarchies based on indented plots – which use indentation to convey information about hierarchy and depth of trees. BarcodeTree [LZD*20] is a novel approach to visualize temporal trees, where the tree at each time step is visualized by a sequence of rectangles representing the nodes. Beck et al. [BWB*14] use a treemap-like visualization called *Icicle Plots* to compare a primary hierarchy to several others. Code Flows [TA08] is a tool to visualize changes in source code repositories by showing movement of code blocks as crossing tubes connecting their position in vertical icicle plots.

Our work is mainly based on two papers that propose temporal treemap visualizations—static two-dimensional visualizations that adopt the visual metaphor of treemaps in one dimension and show the temporal changes in the second (horizontal) dimension. Lukasczyk et al. [LWM*17] have proposed nested tracking graphs, a method initially used to visualize hierarchically nested graphs. Additionally, nested tracking graphs can visualize temporal trees as shown in Figure 1c. However, their layout algorithm can produce many crossings and crossing minimization in this context is an open problem. Köpp and Weinkauff [KW19] introduced the notion of temporal treemaps for nested tracking graphs of temporal trees and have proposed a layout algorithm designed for reducing the crossings. They define a set of constraints that need to be satisfied in order to reduce crossings, and then give implementations of a greedy heuristic and a simulated annealing algorithm that maximize the number of satisfied constraints. However, they do not give a quantitative characterization of crossings in their produced outputs. They also introduced a different variant of visualizing temporal trees that is similar to nested tracking graphs and uses cushion rendering. However, temporal edges are treated differently and crossings can lead to undesirable artifacts. This rendering is thus more suitable when there are few or no crossings. SplitStreams [BNRB21] is another recent visualization similar to temporal treemaps, and we expect that our methods and crossing minimization algorithms are also relevant for minimizing stream crossings in SplitStreams.

Guerra-Gómez et al. [GPPS13] give a taxonomy on types of changes in temporal trees. Temporal treemaps fit partially into their *Type 4*: changes in node weights and topology. Graham and Kennedy [GK10] give a survey on visualizations for multiple (not necessarily temporal) trees. Furthermore, a taxonomy on general dynamic graph visualizations is given by Beck et al. [BBDW17].

Crossing minimization in layered graph drawing. Algorithmic ideas in this paper are inspired by the graph drawing literature for variants of layered graph drawing [HN14]. The well-known *Sugiyama framework* [STT81] is an algorithmic pipeline for drawing directed graphs in an upward layer-by-layer style. A crucial step is to minimize the number of crossings by finding node orders for each layer. Two well-known heuristics are the *barycenter heuristic* [STT81] and the *median heuristic* [EW94]. Ordering problems also appear in different contexts for graph visualization [BBR*16].

Storylines, popularized by an xkcd comic [Mun09], depict the interaction of multiple characters over time. Each character is represented by an x -monotone curve and interactions are depicted by vertical proximity of these curves at specific times. Much focus has been given to drawing storylines with few crossings between

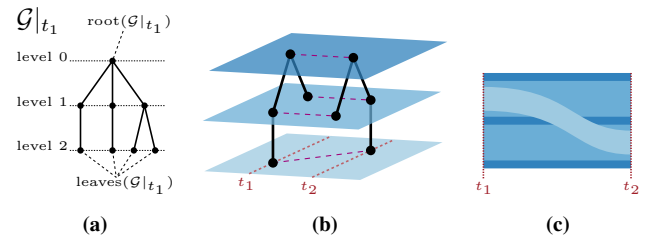


Figure 2: (a) The tree $\mathcal{G}|_{t_1}$ for the temporal tree \mathcal{G} in Figure 1a. (b) A non hierarchy-compliant temporal tree and (c) its visualization as temporal treemap with an unavoidable crossing.

character curves [KNP*15, GJLM16, vDLMW17, vDFF*17]. The integer linear programming model by Gronemann et al. [GJLM16] has inspired the exact algorithms in this paper.

Drawing layered graphs without crossings has appeared in the context of \mathcal{T} -level planarity [ALB*15, JLM98], which is similar to problems discussed in this paper. The problem deals with drawing a layered graph without crossings with additional constraints for the nodes of each layer. Namely, each layer is constrained by a tree whose leaves are the nodes of the layer and leaves that are descendants of the same inner node must appear consecutively. This almost corresponds to our setting, except that in temporal trees temporal edges can also go between non-leaf nodes of the trees.

3. Temporal Trees

We start with some notation needed for the definition of temporal trees. For $n \in \mathbb{N}$, let $[n] = \{1, 2, \dots, n\}$. Let T be a rooted tree. Denote its leaf set as $\text{leaves}(T)$ and for an internal node u , denote its children as $\text{children}(u)$, and by $T(u)$ the subtree rooted at u . Permutations are treated as lists of distinct elements, modifiable by removal, addition, and concatenation of sub-permutations.

Formal Definition. Let $\tau = \{t_1, t_2, \dots, t_\ell\}$ be a set of ℓ time steps, ordered totally by their index. A temporal tree \mathcal{G} is a tuple $\mathcal{G} = (V, E_T, E_N)$ (see Figure 1a). The set V are the nodes, and each node $v \in V$ has a time step $\text{time}(v) \in \tau$. The set E_N consists of directed (from parent to child) hierarchical edges, each connecting two distinct nodes $u, v \in V$ with $\text{time}(u) = \text{time}(v)$. For $t \in \tau$, the graph $\mathcal{G}|_t = (V|_t, E_N|_t)$ consists of all nodes $v \in V$ with $\text{time}(v) = t$, and all edges $(u, v) \in E_N$ such that $\text{time}(u) = \text{time}(v) = t$. For each $t \in \tau$, $\mathcal{G}|_t$ is a rooted tree with a unique root $\text{root}(\mathcal{G}|_t)$ (see Figure 2a for the tree $\mathcal{G}|_{t_1}$ of the temporal tree in Figure 1a). Let v be a node with $\text{time}(v) = t$. We say that v is a leaf if v is a leaf in $\mathcal{G}|_t$. If it is not a leaf, then the set of $\text{children}(v)$ refers to its children in $\mathcal{G}|_t$. The depth or level of a node in a rooted tree is its distance to the root. We define $\text{level}(v)$ of a node $v \in V$ with $\text{time}(v) = t$ as its depth in $\mathcal{G}|_t$. The set E_T consists of directed temporal edges (u, v) , where $\text{time}(u) = t_i$, $\text{time}(v) = t_{i+1}$, and $\text{level}(u) = \text{level}(v)$. Thus, temporal edges connect nodes in the trees of two consecutive time steps, whose level is the same in both trees. Over the course of time, nodes may appear if they have zero incoming temporal edges, disappear if they have zero outgoing temporal edges, split into other nodes if they have more than one outgoing temporal edge, and merge into

a node if they have more than one incoming temporal edge. Additionally, each node $u \in V$ is given a weight $wgt(u) \in \mathbb{R}^+$ associated with it. These values can for example represent sizes of files and directories when the temporal tree represents a file system. We require $wgt(u) \geq \sum_{v \in \text{children}(u)} wgt(v)$.

An edge $(u, v) \in E_T$ is *hierarchy-compliant* if u and v are roots of their respective trees, or the parent of u and the parent of v are also connected in E_T . If all edges in E_T are hierarchy-compliant, then \mathcal{G} is hierarchy-compliant. See, e.g., [Figure 2b](#) for a temporal tree that is not hierarchy-compliant. Köpp and Weinkauff [KW19] only consider hierarchy-compliant temporal trees because edges that are not hierarchy-compliant lead to visual artifacts in temporal tree visualizations, such as unavoidable crossings. Our algorithms also work more generally for temporal trees that are not hierarchy-compliant.

Visualizations and their Combinatorial Characterization. We are concerned with a temporal tree visualization based on the *nested tracking graph* metaphor, termed *temporal treemaps* in this paper. This visualization allows us to define crossings in a natural way; Köpp and Weinkauff [KW19] also use it to verify the crossing-reduction of their simulated annealing algorithm, however, they do not formally define their notion of crossings. Köpp and Weinkauff propose another visualization using cushion maps, which, however, leads to undesirable artifacts whenever crossings appear in the combinatorial characterization. Furthermore, it is hard to capture the term “crossing” for such visualizations. Thus, their cushion map visualization is more appropriate when there are no crossings, which usually happens if there are few merges and splits in the temporal tree. Despite their claim that their visualization is preferable for cases where $wgt(u) = \sum_{v \in \text{children}(u)} wgt(v)$, we will later argue for the efficacy of nested tracking graphs even in such scenarios.

Temporal treemaps draw the temporal edges of the temporal tree. The hierarchical relationships within each time step are implicitly represented by color and containment of temporal edges as in a treemap. The x -axis represents time and each time step corresponds to a vertical strip of the visualization. Temporal edges within a specific layer of the temporal tree have the same color. The vertical width of temporal edges (u, v) is determined by the values $wgt(u)$ and $wgt(v)$. As an example take the temporal tree \mathcal{G} from [Figure 1a](#). A temporal treemap visualization for \mathcal{G} is shown in [Figure 1c](#). Temporal edges are colored in various shades of blue depending on their level – edges in lower levels are coloured in a lighter shade. As stated before, a concern by Köpp and Weinkauff is that such visualizations are “not the best choice” [KW19] for drawing temporal trees where the weight of a node is equal to the sum of weights of its children. Indeed, in the original definition, edges appearing closer to the roots of the temporal tree would be hidden by edges that are further from the roots, as it is also the case in standard treemaps. This can be avoided, though, by adding a small amount of padding at the top and bottom of temporal edges. As correctly observed by Köpp and Weinkauff, such visualizations are not completely faithful anymore, but using small enough padding and proper descriptions of the visualization, this should not impede understanding the data.

Temporal treemaps are combinatorially characterized by a permutation of the leaves of the temporal tree at each time step. The permutation π of the leaves L_t of the tree $\mathcal{G}|_t$ corresponds to the order the leaves L_t appear from top to bottom along the vertical slice

of the temporal treemap corresponding to time step t . These permutations, however, are required to have a specific property: for each internal node u of $\mathcal{G}|_t$, the leaves of the subtree $\mathcal{G}|_t(u)$ rooted at u have to appear consecutively in π to convey the hierarchical containment properties of the temporal tree. We say that π is *restricted by $\mathcal{G}|_t$* . It is not required to explicitly represent the order of internal nodes of $\mathcal{G}|_t$, as this order is extracted from the order of leaves: If two internal nodes u and v in $\mathcal{G}|_t$ are compared, none being a descendant of the other, then we can compare the order of some leaf in $\mathcal{G}|_t(u)$ and some leaf in $\mathcal{G}|_t(v)$. For leaves a, b , we write $a \prec_\pi b$ if a comes before b in π . For two nodes u, v in $\mathcal{G}|_t$, not necessarily leaves and neither u being a descendant of v nor v being a descendant of u , we write $u \prec_\pi v$ if leaves in $\mathcal{G}|_t(u)$ come before leaves in $\mathcal{G}|_t(v)$. Now, let $\mathcal{G} = (V, E_T, E_N)$ be a temporal tree on a set of time steps $\tau = \{t_1, t_2, \dots, t_\ell\}$. A *combinatorial layout* of \mathcal{G} is a sequence of permutations $(\pi_1, \pi_2, \dots, \pi_\ell)$ such that π_i is a permutation of the leaves of the tree $\mathcal{G}|_{t_i}$ and is restricted by $\mathcal{G}|_{t_i}$. Each combinatorial layout corresponds to a drawing of \mathcal{G} as a temporal treemap.

4. Formalizing Crossings

Let us now propose two natural definitions of crossings in temporal tree layouts that can be easily checked for combinatorial layouts of temporal trees. Generally, we will define what it means that two temporal edges $e = (u, v)$ and $f = (u', v')$ cross. First, it is clear that for a crossing to happen e and f must be edges between the same two time steps t_i and t_{i+1} . Second, the order of u and u' in π_i must be different to the order of v and v' in π_{i+1} . So for a crossing to happen we either have $u \prec_{\pi_i} u'$ and $v' \prec_{\pi_{i+1}} v$, or $u' \prec_{\pi_i} u$ and $v \prec_{\pi_{i+1}} v'$. Note that for this to be well-defined, neither u and u' nor v and v' can be in an ancestor-descendant relation.

We now give a formal definition of crossing, one of its flaws, and a second definition that overcomes this.

Crossing Definition 1. Given a combinatorial layout $\mathcal{L} = (\pi_1, \pi_2, \dots, \pi_\ell)$ of a temporal tree, we say that two temporal edges $e = (u, v)$ and $f = (u', v')$ cross if

1. there exists an $i \in \{1, 2, \dots, \ell - 1\}$ such that $\text{time}(u) = \text{time}(u') = t_i$ and $\text{time}(v) = \text{time}(v') = t_{i+1}$,
2. neither u, u' , nor v, v' are in ancestor-descendant relation, and
3. $u \prec_{\pi_i} u'$ and $v' \prec_{\pi_{i+1}} v$, or $u' \prec_{\pi_i} u$ and $v \prec_{\pi_{i+1}} v'$.

Let $cr(\mathcal{L})$ be the number of edge-pairs that cross in the combinatorial layout \mathcal{L} . We deal with the following optimization problem.

Problem 1 Given a temporal tree \mathcal{G} , compute a combinatorial layout \mathcal{L} of \mathcal{G} with the minimum number of crossings $cr(\mathcal{L})$.

However, with this definition of crossing we can have many crossings for a combinatorial layout whose drawing as temporal treemap does not visually appear to have as many crossings. See [Figure 3](#) for an example with 9 crossings according to the above definition, but the drawing appears to only have one “nested” crossing.

Crossing Definition 2. We overcome the problem in [Figure 3](#) with a second crossing definition. Intuitively, it makes sense to require for a crossing that at least one endpoint of both edges involved has to be a leaf in its tree. This is not enough as can be observed for the

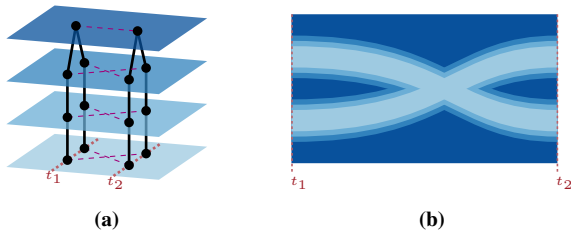


Figure 3: A temporal tree (a) and one of its drawings as temporal treemap (b) such that $cr(\mathcal{L}) = 9$ and $cr_L(\mathcal{L}) = 1$ for the corresponding combinatorial layout.

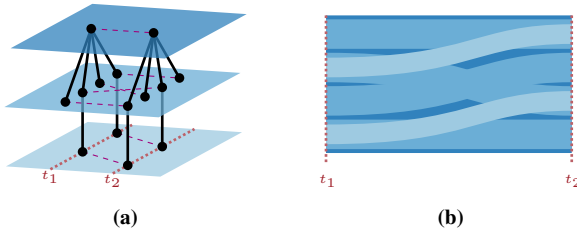


Figure 4: A temporal tree (a) and a drawing as temporal treemap (b) such that one of the edges on level one involved in a crossing has two non-leaf endpoints.

level-1 crossing in Figure 4, where an edge with two non-leaf endpoints is involved in a crossing which would not have been counted otherwise. But these two edges have specific properties. Namely, we say that a temporal edge (u, v) between $\mathcal{G}|_{t_i}$ and $\mathcal{G}|_{t_{i+1}}$ is a *leaf-edge* if there is no temporal edge (u', v') such that u' is a descendant of u in $\mathcal{G}|_{t_i}$ and v' is a descendant of v in $\mathcal{G}|_{t_{i+1}}$. This leads to the second crossing definition. We say that two temporal edges e and f form a *leaf crossing* in a combinatorial layout \mathcal{L} if they obey all properties of Crossing Definition 1, and

4. e and f are both leaf-edges.

Let $cr_L(\mathcal{L})$ be the number of leaf crossing in \mathcal{L} . Again, we define a computational problem.

Problem 2 Given a temporal tree \mathcal{G} , compute a combinatorial layout \mathcal{L} of \mathcal{G} with the minimum number of leaf crossings $cr_L(\mathcal{L})$.

Note that the crossing in Figure 3 is not the same as a crossing between only two single temporal edges. Thus, it might make sense to penalize such a “nested” crossing more and Crossing Definition 1 is also a valid option. We propose algorithms for minimizing both types of crossings in the next section.

Above, we have ignored crossings stemming from non hierarchy-compliant trees. See Figure 2 for an example, where the level-2 leaf edge crosses the border of its parents. Such crossings cannot be avoided and they appear in every combinatorial layout of the input tree. We thus ignore them in our definition of crossings, as we will never be able to remove such crossings.

5. Algorithms for Minimizing Crossings

In this section, we will propose algorithms for solving Problem 1 and Problem 2. First we will show that both problems are NP-hard.

This justifies the investigation of heuristics and non-polynomial exact approaches. Then we will discuss the algorithms by Köpp and Weinkauff. Lastly we will present our new algorithms—integer linear programs computing optimal solutions, and two heuristics.

5.1. Complexity

NP-hardness. NP-hardness follows by a simple reduction from the NP-hard problem BIPARTITE CROSSING NUMBER [GJ83]. The problem input is a tuple (G, k) , where G is a bipartite graph with the node set V partitioned into V_1 and V_2 , and k is an integer. The problem asks whether there is a drawing of the graph with edges as straight lines, with the nodes on two parallel horizontal lines such that all nodes in V_1 are placed on the top line, all nodes in V_2 are placed on the bottom line, and the drawing has at most k crossings. This problem can be simulated with a temporal tree \mathcal{G} for two time steps t_1, t_2 , such that

- $\mathcal{G}|_{t_1}$ is a single root with an edge to every node in V_1 ,
- $\mathcal{G}|_{t_2}$ is a single root with an edge to every node in V_2 , and
- E_T consists of the edges in G plus an edge connecting the roots of $\mathcal{G}|_{t_1}$ and $\mathcal{G}|_{t_2}$.

As \mathcal{G} only has two levels, crossings and leaf crossings are the same for every combinatorial layout. Also, it is easy to see that \mathcal{G} has a combinatorial layout with at most k crossings if and only if (G, k) is a yes-instance of BIPARTITE CROSSING NUMBER. This is the case as there is essentially a one-to-one correspondence between such drawings of G and the temporal treemaps corresponding to combinatorial layouts of \mathcal{G} . Thus, the following holds.

Theorem 5.1 Problem 1 and Problem 2 are NP-hard.

Additionally, we show in the supplementary material [DN23] that planar temporal treemaps – those which have a layout with zero crossings – can be recognized in polynomial time.

5.2. The Approach of Köpp and Weinkauff

Let us now discuss the algorithms of Köpp and Weinkauff [KW19] to compute combinatorial layouts with few crossings. They compute a single permutation π of the union of leaves of trees for each time step, that is, $\cup_{t \in \tau} \text{leaves}(\mathcal{G}|_t)$. The permutations of leaves for a single time step can then be obtained by removing all other leaves from that permutation. Their heuristic algorithms try to compute such a permutation maximizing the number of satisfied *hierarchy* and *topology* constraints imposed by the temporal tree. Essentially, these constraints enforce specific leaves to be together in the permutation π , and the constraints should ensure that the resulting combinatorial layout has few crossings. They propose two algorithms for this. A heuristic that takes a single constraint, satisfies it by placing the involved leaves together in π as long as the number of violated constraints decreases. The second algorithm is a simulated annealing algorithm that works similarly as the heuristic, but sometimes accepts worse solutions based on a temperature function that decreases over time. Furthermore, the algorithms of Köpp and Weinkauff work for an aggregated variant of the temporal tree that aggregates sets of nodes with specific properties into single nodes. The order then only needs to be computed for the fewer aggregated nodes. The aggregated tree needs less space which improves the

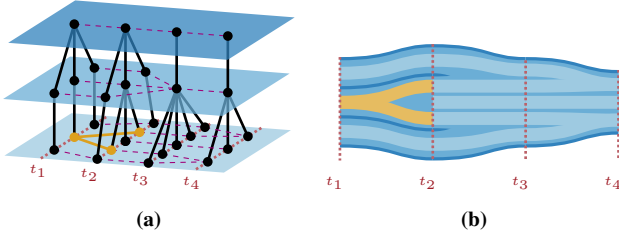


Figure 5: A temporal tree (a) and one of its drawings as temporal treemaps (b) without crossings. A set of leaves involved in a topology constraint are marked in orange.

runtime of the heuristic and simulated annealing algorithm. Details about the aggregation can be found in [KW19]. In our algorithms we always work with non-aggregated temporal trees, as we do not see how to integrate aggregated variants into our algorithms, and as they sometimes do not preserve the number of crossings between aggregated and non-aggregated trees.

We now discuss some concerns with these approaches, and how they can lead to more crossings compared to other algorithms. First, it can happen that after completion of an algorithm, some hierarchy constraints are not satisfied and need to be repaired as these constraints reflect the hierarchical relationships of the single trees \mathcal{G}_t for time steps t . This is certainly undesirable, and can be prevented by allowing only permutations where all hierarchy constraints are satisfied; such permutations always exist and can be easily computed. Second, there are temporal trees where crossings can be avoided, even if there exists no permutation of the leaves satisfying all hierarchy and topology constraints. Such a temporal tree is given in Figure 5. The combinatorial layout in Figure 5 has zero crossings. However, the corresponding combinatorial layout does not satisfy a topology constraint imposed by the construction of Köpp and Weinkauff which would force all orange leaves to appear together in π . In fact, there exists no combinatorial layout for this temporal tree that satisfies all hierarchy and topology constraints as they are defined by Köpp and Weinkauff. Our algorithms can handle such inputs and produce a combinatorial layout with zero crossings. Lastly, we will see in our experiments that the simulated annealing algorithm is very slow as it requires a lot of iterations and each iteration has to update the entire permutation.

5.3. Integer Linear Programming

Our first new algorithm uses the integer linear programming (ILP) technique to compute optimal solutions (if given enough time) despite the NP-hardness of the crossing minimization problems. It builds on ideas from previous ILP formulations for a similar problem in storyline visualizations [GJLM16].

For each time step $t_i \in \tau$ and each two leaves $u, v \in \text{leaves}(\mathcal{G}|_{t_i})$, $u \neq v$, we have a binary ordering variable $x_{u,v}$. This variable will be 1 if and only if $u \prec_{\pi_i} v$ in the solution. Let E_{cr}^2 contain all unordered pairs $\{e, f\}$ of temporal edges that could cross in some combinatorial layout of \mathcal{G} . This set consists of all pairs of temporal edges e, f between the same two time steps such that neither the sources nor the targets are in an ancestor-descendant relation in the correspond-

ing trees. For each $\{e, f\} \in E_{cr}^2$ we have a binary crossing variable $y_{e,f}$ that is 1 if and only if e and f cross in the solution.

ILP for minimizing crossings. We start by describing the objective and constraints for the ILP that minimizes crossings. Below, u, v, w are always assumed to be pairwise different. For a temporal edge $e = (u, v)$ between time steps t_i and t_{i+1} , let $\text{src}(e)$ be any leaf in $\mathcal{G}|_{t_i}(u)$, and let $\text{tgt}(e)$ be any leaf in $\mathcal{G}|_{t_{i+1}}(v)$.

$$\text{minimize: } \sum_{\{e,f\} \in E_{cr}^2} y_{e,f} \quad (1)$$

$$\text{subject to: } x_{u,v} + x_{v,u} = 1, i \in [\ell], u, v \in \text{leaves}(\mathcal{G}|_{t_i}) \quad (2)$$

$$x_{u,v} + x_{v,w} + x_{u,w} \leq 2, i \in [\ell], u, v, w \in \text{leaves}(\mathcal{G}|_{t_i}) \quad (3)$$

$$x_{\text{src}(e), \text{src}(f)} - x_{\text{tgt}(e), \text{tgt}(f)} \leq y_{e,f}, \{e, f\} \in E_{cr}^2 \quad (4)$$

$$x_{\text{tgt}(e), \text{tgt}(f)} - x_{\text{src}(e), \text{src}(f)} \leq y_{e,f}, \{e, f\} \in E_{cr}^2 \quad (5)$$

The objective (1) minimizes the number of crossings. The constraint (2) ensures antisymmetry of the computed permutations, while (3) ensures transitivity (see e.g. [GJR85]). The constraints (4) and (5) ensure that the crossing variable $y_{e,f}$ is one whenever the two involved edges e and f cross in the resulting combinatorial layout. However, this is still not enough for the resulting permutations to satisfy the hierarchy constraints imposed by the structures of the trees for each time step. For this we have to introduce tree constraints [GJLM16] that ensure that leaves in a subtree rooted at an internal node appear consecutively in the corresponding permutation. Thus, for each $t_i \in \tau$, each non-root internal node $u \in V(\mathcal{G}|_{t_i})$, each pair v, v' of leaves in $\mathcal{G}|_{t_i}(u)$, and each w that is a leaf in $\mathcal{G}|_{t_i}$ and is not in $\mathcal{G}|_{t_i}(u)$ we add the following constraint.

$$x_{v,w} = x_{v',w} \quad (6)$$

If all constraints of type (6) are satisfied then either w is either before or after all leaves of $\mathcal{G}|_{t_i}(u)$ in π_i , but never in between. The permutations $\pi_1, \pi_2, \dots, \pi_\ell$ can then be extracted from the ordering variables in the solution computed by any standard ILP solver. Due to the objective corresponding to the number of crossings, any optimal solution will correspond to a combinatorial layout with the minimal number of crossings. In our implementation we additionally remove the symmetries created by ordering variables. That is, for distinct $u, v \in \text{leaves}(\mathcal{G}|_{t_i})$ we only have one of two variables $x_{u,v}, x_{v,u}$ in the model and adjust the constraints accordingly.

ILP for minimizing leaf crossings. The above formulation can be converted to minimize leaf crossings instead of crossings. Let $E_{Lcr}^2 = E_{cr}^2 \cap (E_L \times E_L)$ be the set of all pairs of leaf edges that can cross. Instead of having a crossing variable for each pair of edges in E_{cr}^2 , we now have a variable $y_{e,f}$ for each pair $\{e, f\} \in E_{Lcr}^2$. The ILP model can be obtained from the above formulation by replacing every occurrence of E_{cr}^2 by E_{Lcr}^2 . The combinatorial layout that minimizes leaf crossings can again be extracted from the ordering variables in an optimum solution of the ILP model.

5.4. Barycenter and Median Heuristic

Since solving ILPs is not guaranteed to terminate within feasible amount of time for large instances, we also present two heuristic algorithms. These algorithms do not guarantee to produce optimal

Algorithm 1: First forward iteration of the barycenter heuristic.

Input: Temporal tree \mathcal{G} on time steps t_1, t_2, \dots, t_ℓ
Output: An order π_i for each $i = 1, 2, \dots, \ell$.

```

1 for  $i = 1, 2, 3, \dots, \ell$  do
2   if  $i = 1$  then  $\pi_1 \leftarrow$  random order of leaves( $\mathcal{G}_{t_1}$ );
3   else
4     pred_positions( $u$ )  $\leftarrow$  () for all nodes  $u$  in  $\mathcal{G}|_{t_i}$ ;
5      $\pi_i \leftarrow$  computeOrder(root( $\mathcal{G}|_{t_i}$ ));
6     compute pos( $v$ ) for all nodes  $v$  in  $\mathcal{G}|_{t_i}$  with respect to  $\pi_i$ ;
7 return ( $\pi_1, \pi_2, \dots, \pi_\ell$ );

```

Algorithm 2: Recursive computation of the permutation of leaves of the subtree rooted at u .

```

1 Function computeOrder( $u$ ):
2   if  $u$  is a leaf then
3     order  $\leftarrow$  ( $u$ )
4   else
5     for  $v$  in children( $u$ ) do
6        $\pi_v \leftarrow$  computeOrder( $v$ );
7       pred_positions( $u$ ).append(pred_positions( $v$ ));
8     order  $\leftarrow$  ();
9     for  $v$  in children( $u$ ) sorted by
10      average(pred_positions( $v$ )) do
11       order.append( $\pi_v$ );
12   for temporal edge  $(v, w) \in E_T$  such that  $w = u$  do
13     pred_positions( $u$ ).append(pos( $v$ ));
14   return order

```

solutions, but will compute good solutions very fast. Each algorithm is based on a respective algorithm for layered graph drawing [STT81, EW94]. Layered graph drawing is concerned with drawing a graph on ℓ layers, each node being assigned to a specific layer and edges connecting nodes of consecutive layers. In the well-known Sugiyama-framework [STT81] a key step is to order nodes on the respective layers to minimize the number of crossings. This is done by multiple passes up and down the layers. In each pass up, the permutation of nodes in layer i is computed based on the permutation of nodes in layer $i - 1$ with i going from 2 to ℓ . In each pass down the role of i and $i - 1$ are reversed and i goes from ℓ to 2. Two well-known strategies for adjusting the positions of nodes in layer i are the *barycenter heuristic* [STT81] and the *median heuristic* [EW94]. The barycenter heuristic assigns a node in layer i the average position of its adjacent nodes in layer $i - 1$, while the median heuristic uses the median instead of the average.

We adjust the median and barycenter heuristic to compute permutations of leaves of temporal trees. Here we also have to consider temporal edges between non-leaf nodes, and we have to compute permutations adhering to the hierarchical structure of the trees of each time step. We start by describing the barycenter heuristic and later describe the changes required for the median heuristic.

Barycenter heuristic. Pseudocode for the first iteration of the barycenter heuristic is shown in Algorithm 1. The algorithm computes orders $\pi_1, \pi_2, \dots, \pi_\ell$ of the combinatorial layout in this order. Further, given the already computed order π_i it computes the auxiliary values $\text{pos}(u)$ for all nodes u in \mathcal{G}_{t_i} which are defined as follows. If u is a leaf then $\text{pos}(u)$ is the index of u in π_i . Otherwise, $\text{pos}(u)$ is recursively computed as $\text{pos}(u) = \min_{v \in \text{children}(u)} \text{pos}(v)$.

The permutation π_1 is set to a random order of the leaves in \mathcal{G}_{t_1} . The recursive computation of the permutation π_i for $i \geq 2$ given π_{i-1} is shown in Algorithm 2. The function `computeOrder(u)` computes a permutation for all leaves in $\mathcal{G}|_{t_i}(u)$. So, π_i is computed by applying the function to the root of $\mathcal{G}|_{t_i}$. The auxiliary lists `pred_positions(u)` (*predecessor-positions*) store a list of values $\text{pos}(v)$ for each temporal edge going from some v in $\mathcal{G}|_{t_{i-1}}$ to a node in the subtree $\mathcal{G}|_{t_i}(u)$. These lists are used to determine the order between children of u . The function `computeOrder(u)` recursively computes such a permutation as follows. If u is a leaf, then the permutation consists of the single leaf u . Otherwise, for each child v of u the order π_v is computed recursively. These are then concatenated by ascending order of the average value in `pred_positions(v)`. That is, if v_1, v_2, \dots, v_p are the children of u sorted by the average value in their corresponding `pred_positions` lists, then the computed order is the concatenation of $\pi_{v_1}, \pi_{v_2}, \dots, \pi_{v_p}$ in this order. Lastly, the function `computeOrder` also computes the values `pred_positions(u)` by adding all values `pred_positions(v)` for children v of u and adding values $\text{pos}(v)$ for temporal predecessors v of u . Temporal predecessors are nodes v in $\mathcal{G}|_{t_{i-1}}$ such that there exists a temporal edge (v, u) .

Algorithm 1 is what we call a *forward iteration*. A backward iteration instead updates the orders $\pi_{\ell-1}, \pi_{\ell-2}, \dots, \pi_1$ in this order. For example, when computing π_i based on π_{i+1} we replace temporal predecessors of u by *temporal successors* of u in the above description. These are nodes v in $\mathcal{G}|_{t_{i+1}}$ such that there exists a temporal edge (u, v) . The complete barycenter heuristic consists of $s \in \mathbb{N}$ sweeps; each sweep first does a forward iteration and then does a backward iteration. Only the first forward iteration needs to initialize the random order π_1 in line 2, as π_1 is already initialized in later iterations. Later we discuss how the value s influences the results.

Median heuristic and implementation details. The barycenter heuristic can very easily be transformed into the median heuristic. We simply replace the average in line 9 of Algorithm 2 by the median, i.e., we take `median(pred_positions(v))` instead of `average(pred_positions(v))` to sort the children v of u . The complete median heuristic is performed as above by doing s sweeps of forward and backwards iterations. In our implementation we make some optimizations to Algorithm 2. For the barycenter heuristic, instead of storing `pred_positions(u)` as an array we only store the sum and size of this list. The average is then easily computable. For the median heuristic we store `pred_positions(u)` in a sorted multiset. In line 9 of Algorithm 2, initially the nodes u with non-empty lists `pred_positions(u)` are sorted by average/median within their respective lists. Next, the remaining nodes are sorted into this order with comparisons between pairs of nodes – if at least one has an empty `pred_positions` list – relying on the pre-established order π_i (during the first forward iteration, where π_i is not yet known, they are simply appended at the end).

Adaptation for leaf crossings. The above barycenter and median heuristics are designed for minimizing crossings. A slight adaptation can be used for minimizing leaf crossings: Instead of iterating over E_T in line 11, we only iterate over all leaf edges. This makes sure that only the edges that can be involved in leaf crossings are used to determine the computed permutations.

6. Experiments

We performed several experiments comparing the performance of all our algorithms and the algorithms of Köpp and Weinkauff [KW19] with respect to the number of (leaf) crossings and runtime. We first present the setup, dataset, and then present the results.

6.1. Setup

Implementation. We implemented seven different algorithms, as described in Section 5.2–Section 5.4. They are labelled as follows:

KW Heuristic+annealing algorithm by Köpp and Weinkauff. We use the implementations by Köpp and Weinkauff working with the aggregated variant of temporal trees.

ILP ILP formulation for crossings.

ILP-LE ILP formulation for leaf crossings.

BARY Barycenter heuristic for crossings.

BARY-LE Barycenter heuristic for leaf crossings.

MEDI Median heuristic for crossings.

MEDI-LE Median heuristic for leaf crossings.

The algorithm KW is a combination of the heuristic and simulated annealing. First the heuristic is applied for twice as many iterations as there are topology and hierarchy constraints. Then the simulated annealing algorithm takes the output of the heuristic as input and is executed with initial temperature 5, 10 iterations per temperature, temperature decay 0.9, and minimum temperature 10^{-5} . These choices give a good tradeoff between execution time of the algorithm and its performance, as also determined by Köpp and Weinkauff [KW19]. The number of sweeps for all the barycenter and median heuristic variants was set to $s = 10$; we determined experimentally that this value produces good results. In fact, already one sweep produced good results, and the average number of crossings reduced by less than 5% when using ten sweeps instead of one.

All algorithms are implemented in C++17 and compiled with g++ version 11.4.0 using the compile flags `-Wall -Wextra -O3`. We make use of a data structure implementation for temporal trees by Köpp and Weinkauff [KW19]. The integer linear programming formulations were solved by the Gurobi solver [Gur23] with version 10.0.0 using its C++ interface. All executions were performed on a cluster of 48 nodes, each containing 2x AMD EPYC 7402, 2.80GHz 24-core processors running Ubuntu 18.04.6 LTS. The memory limit was set to 8GB. As each individual execution was limited to a single thread, the configuration is comparable to the hardware of an ordinary user machine. The timeout of a single execution of an algorithm for an instance was set to 10 minutes.

Test data. We have a set of hierarchy-compliant instances already considered by Köpp and Weinkauff [KW19]. The *Viscous Fingers* data set originally stemming from the IEEE visualization contest

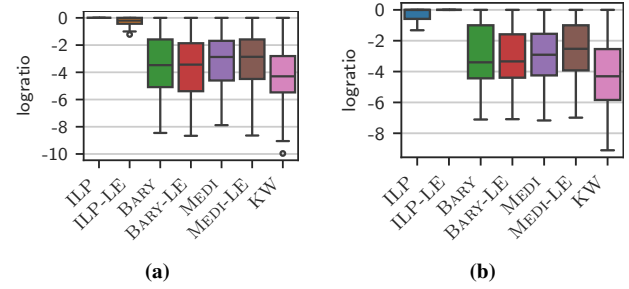


Figure 6: Boxplots of the logratios of (a) crossings (b) leaf crossings for all algorithms. We consider all instances where (a) ILP (b) ILP-LE neither times out nor runs into memory limit. The horizontal lines of the boxes give the first quartile, median, and third quartile from bottom to top. The whiskers show the minimum and maximum value without outliers.

[GG16] consists of 919 nodes and 61 time steps, and represents a process in fluid dynamics. The *Cylinder* data set describes vortex activity in the wake of a square cylinder by means of the Okubo-Weiss criterion [CBI*05, vFWTS08]. It consists of 28904 nodes and 508 time steps. Furthermore, we split the *Cylinder* dataset into small instances by taking a range of time steps of the tree. Namely, for each $k \in \{20, 50, 100, 150, 200, 300\}$, we created instances taking the *Cylinder* dataset over the time step range $[20 \cdot i + 1, 20 \cdot i + k]$ for each integer i such that the range is a subset of $[1, 508]$. Overall, this results in a set of 114 benchmark instances of varying sizes. We ran each algorithm on each of the 114 temporal trees once.

6.2. Results

Crossings. Here we consider the number of crossings obtained by each algorithm. Algorithm ILP computes the optimum value. So, for every other algorithm ALG we compute for each instance the logratio $\log_2((cr_{opt} + 1)/(cr_{ALG} + 1))$. Here, cr_{ALG} is the number of crossings obtained by ALG and cr_{opt} is the minimum number of crossings obtained by ILP. The additive one is because we want to avoid undefined values when one of cr_{opt} or cr_{ALG} is zero. Higher values of the logratio indicate better performance of the algorithm and the logratio is never positive. A logratio of $-k$ means that the algorithm produces at least 2^k as many crossings as the optimum. If, e.g., $k = -2$ then there are 4 times as many crossings as in the optimal solution. For all plots involving logratios we only considered instances where ILP neither timed out nor ran into a memory limit (otherwise we do not know the optimum values).

Figure 6a depicts the logratios in a boxplot for each algorithm over 69 instances. Note that the logratio of ILP is always the optimal value zero. It is interesting that even though ILP-LE is designed to minimize leaf crossings, it performs very well w.r.t. crossings, at least when compared to the other heuristic algorithms. Furthermore, all variants of barycenter and median heuristics perform very similar, the median heuristic variants performing a bit better. Again, variants for leaf crossings perform very similar to its counterparts designed for crossings. Lastly, the algorithm KW performs worse than the remaining algorithms by a factor of at least two.

In Figure 7 we give a detailed comparison between the previous

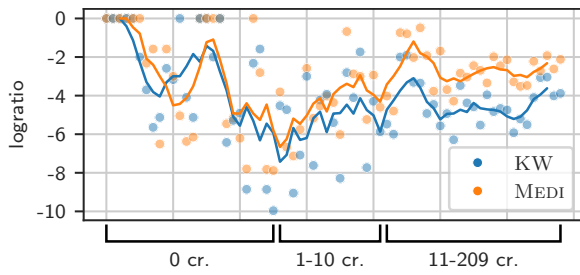


Figure 7: Comparison of the algorithms MEDI and KW w.r.t. to the number of crossings. Each point corresponds to an instance and an algorithm. The instances are ordered on the x -axis equidistant by their optimum number of crossings. The lines correspond to a running average of 5 points.

algorithm KW and one of our heuristics, MEDI. Due to their similar performance this serves as a representative comparison to KW for all our heuristics. The y -axis corresponds to the logratio obtained by an algorithm. The instances are ordered along the x -axis by their optimum number of crossings, ties are broken by going in increasing order of the number of leaves. Ranges of the minimum number of crossings are shown on the x -axis of the plot. Each dot corresponds to an algorithm and an instance, the lines are running averages of 5 points. MEDI consistently outperforms KW for nearly all instances. Still, both algorithms are far from the optimum, even when the optimum number of crossings is zero. Overall, the average logratio for KW is ≈ -4.1 , while it is ≈ -3.1 for MEDI. So MEDI produces on average less than 50% of the crossings of KW.

Leaf crossings. We continue with results for leaf crossings. We use ILP-LE to compute optimum solutions and consider again the logratio $\log_2((lecr_{opt} + 1)/(lecr_{ALG} + 1))$ where $lecr$ refers to the number of leaf crossings produced by an algorithm. For the same reasons as before, we only consider instances where ILP-LE neither timed out nor ran into a memory limit.

Figure 6b shows a boxplot of these logratios for each algorithm over 68 instances. Again, ILP performs well even though it is not designed to minimize leaf crossings. The barycenter and median heuristic variants perform better than KW. MEDI-LE is slightly better than the other heuristic variants. In Figure 8, MEDI-LE and KW are compared in detail. The instances are now sorted along the x -axis by their optimum number of leaf crossings, ties are again broken by the number of leaves in increasing order. The plot was constructed the same way as Figure 7. MEDI-LE outperforms KW in nearly all the considered instances. The average logratio for KW is ≈ -4.2 , while it is ≈ -2.7 for MEDI-LE. This means the number of leaf crossings are less by a factor of about 3 for MEDI-LE.

Computational Scalability. Lastly, we investigate how the algorithms perform for larger instances. We measured the runtime in milliseconds of each algorithm for each individual instance. Out of the 114 instances ILP and ILP-LE exceeded the memory limit twice each and produced 45 and 46 timeouts, respectively. The algorithm KW had 24 timeouts, while our heuristics computed solutions in under one second for every instance. Figure 9 shows a

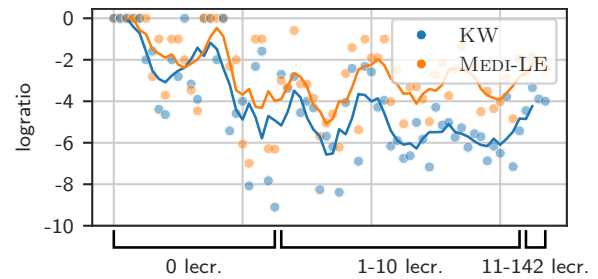


Figure 8: Comparison of the algorithms MEDI-LE and KW w.r.t. to the number of leaf crossings. Each point corresponds to an instance and an algorithm. The instances are ordered on the x -axis equidistant by their optimum number of leaf crossings. The lines correspond to a running average of 5 points.

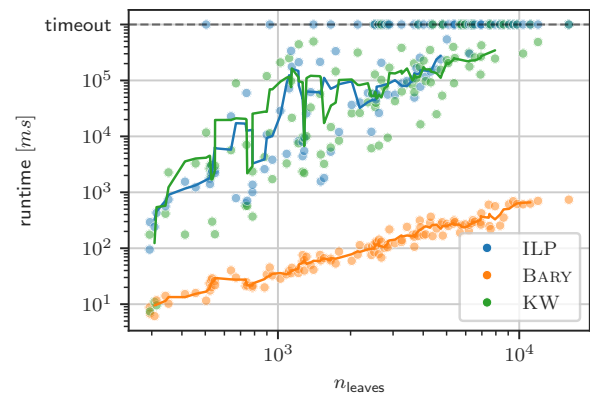


Figure 9: Scatter plot of runtimes for selected algorithms. The lines correspond to a running average of five non-timed out instances and both axes are logarithmic.

scatter plot of runtimes by the total number of leaves in the temporal tree instance. The runtimes of ILP-LE are very similar to ILP, so we only show results for ILP. All our heuristic algorithms are nearly equal w.r.t. runtime, so we only show results for BARY. Timeouts are depicted on the topmost horizontal line. To spread out the scattered points, the x - and y -axes are scaled logarithmically. It is clearly visible that BARY is much faster than both other shown algorithms, its runtime growing linearly with the number of leaves in the instance. Even for over 10,000 leaves the runtime is still below one second. The runtimes of ILP and KW follow a more chaotic behavior. For ILP this is because it computes optimal solutions for an NP-hard problem. It is quite surprising for KW, however, as we would have expected the runtime to scale linearly with the number of leaves. This could be because KW works with an aggregated variant of the temporal tree whose size does not exactly correspond to the number of leaves in the instance. If we consider the ratio between runtimes of KW and BARY, then the median is ≈ 324 and the average is ≈ 921 . So despite both being heuristics, we observe that BARY computes solutions with significantly fewer (leaf) crossings by a factor of 2–3 on average and is at the same time significantly faster by 2–3 orders of magnitude. Even though ILP always computes optimal solutions, its runtimes are often comparable to KW.

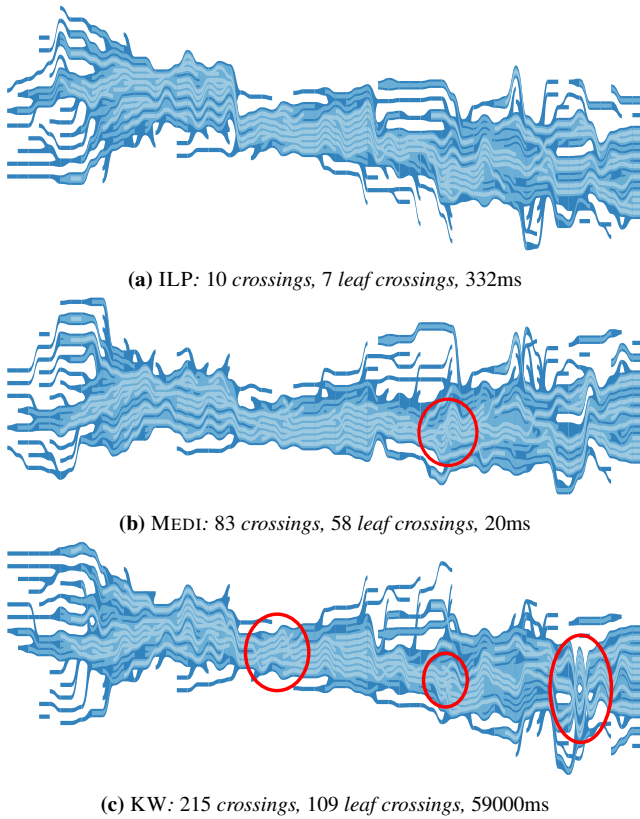


Figure 10: Combinatorial layouts computed for the Viscous Fingers dataset by different algorithms shown as temporal treemaps. Captions report number of crossings, leaf crossings, and runtime. The input temporal tree has 939 nodes, 670 leaves, and depth 3.

7. Application Example

Finally, we show how the different numbers of crossings produced by the algorithms affect the actual representation as temporal treemap for a real-world data set. For this, we chose the Viscous Fingers data set [GG16] that was used before to show temporal treemaps [KW19] and removed the node weights to better recognize crossings. We compare the combinatorial layout produced by the algorithms ILP, KW, and MEDI in Figure 10. The layout computed by KW has 20 times as many crossings as the ILP layout and this is also clearly visible, e.g., in the regions with many crossings encircled in red. Furthermore, ILP was able to compute the optimal layout of this instance in only 332ms, which is almost 200 times faster than the runtime of KW. MEDI computed the layout in only 20ms, and has half the crossings of KW but still 8 times more than ILP. More examples are in the supplementary material [DN23].

8. Discussion and Limitations

We have defined two types of relevant crossings for temporal tree visualizations and conjecture that minimizing each of them leads to more readable temporal tree visualizations. We conjecture that a significant reduction in the number of crossings as observed for ILP in Section 7 also improves readability of the tem-

poral treemaps. But this will have to be verified by a separate user study, which is beyond the scope of this paper. However, Köpp and Weinkauff already argued strongly for minimizing crossings in their paper [KW19] and empirical research in general graph drawing has shown that crossing minimization is among the most influential aspects of improving graph readability [Pur97, WPCM02]. Finally, it is interesting to expand the crossing minimization approaches to other systems such as Splitstreams [BNRB21].

A restriction of temporal treemaps is that temporal edges cannot connect nodes from different levels. It is yet open how to adapt temporal treemaps for these cases. However, such temporal edges can be simulated by letting the first node disappear and another appear. Our algorithms can be adapted for such edges; it is only the visualizations that impose this restriction. Furthermore the visual scalability of temporal treemaps is restricted by the number of perceivable color shades which are assigned to the different levels.

It has yet to be determined which variant of crossings is more effective, crossings or leaf crossings. Again, user studies comparing both need to be conducted. However, since we provide algorithms for both variants, the appropriate one is ready once it is known which variant performs better. It also makes sense to study a combination of both: crossings “overcount” crossings such as in Figure 3; leaf crossings might “undercount” the crossings in Figure 3 as by the definition of leaf crossing it is equivalent to only two leaf edges crossing each other in a temporal tree of depth one. Hence, there is a trade-off between both and a combination might be of interest.

Our heuristics compute better solutions than previous heuristics from the literature and are much faster. However, solutions still have significantly more crossings than the optimal solutions computed by the ILP approaches. Further improvements of our heuristics can be studied, or one could try to adapt alternative algorithms for layered graph drawing to temporal trees, for example the sifting technique [MSM99]. In fact, our ILP algorithms were able to compute optimal solutions within 10 minutes for all temporal trees with up to 500 leaves in our test data set, corresponding to medium-size real world data, and are thus the recommended choice for optimizing the layout of such trees. In interaction settings, or where an efficient ILP solver is not available, or in instances with more than 500 leaves our heuristics are to be preferred.

Lastly, we have performed experiments for a relatively small set of established real-world instances, one of which was split into a larger set of instances. A larger set of instances would have been desirable, however, real-world temporal tree data are not as abundant as other types of graphs and we use a superset of the instances used by Köpp and Weinkauff [KW19]. Such instances have to contain sufficient changes in the tree over time to lead to crossings.

Acknowledgement

This work is supported by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT19035]. The authors acknowledge TU Wien Bibliothek for financial support through its Open Access Funding Programme.

References

- [ALB*15] ANGELINI P., LOZZO G. D., BATTISTA G. D., FRATI F., ROSELLI V.: The importance of being proper: (in clustered-level planarity and T-level planarity). *Theor. Comput. Sci.* 571 (2015), 1–9. doi:10.1016/j.tcs.2014.12.019. 3
- [AMST23] AIGNER W., MIKSCH S., SCHUMANN H., TOMINSKI C.: *Visualization of Time-Oriented Data*, 2nd ed. Springer, 2023. 2
- [BBDW17] BECK F., BURCH M., DIEHL S., WEISKOPF D.: A taxonomy and survey of dynamic graph visualization. *Comput. Graph. Forum* 36, 1 (2017), 133–159. doi:10.1111/CGF.12791. 1, 3
- [BBLW14] BURCH M., BLASCHECK T., LOUKA C., WEISKOPF D.: Visualizing hierarchy changes by dynamic indented plots. In *Proc. IEEE Symposium on Information Visualization (InfoVis'14)* (2014), Laramée R. S., Kerren A., Braz J., (Eds.), SciTePress, pp. 91–98. doi:10.5220/0004652400910098. 3
- [BBR*16] BEHRISCH M., BACH B., RICHE N. H., SCHRECK T., FEKETE J.: Matrix reordering methods for table and network visualization. *Comput. Graph. Forum* 35, 3 (2016), 693–716. doi:10.1111/CGF.12935. 3
- [BD05] BALZER M., DEUSSEN O.: Voronoi treemaps. In *Proc. IEEE Symposium on Information Visualization (InfoVis'05)* (2005), Stasko J. T., Ward M. O., (Eds.), IEEE Computer Society, pp. 49–56. doi:10.1109/INFVIS.2005.1532128. 2
- [BH16] BARTOLOMEO M. D., HU Y.: There is more to streamgraphs than movies: Better aesthetics via ordering and lassoing. *Comput. Graph. Forum* 35, 3 (2016), 341–350. doi:10.1111/CGF.12910. 2
- [BHvW00] BRULS M., HUIZING K., VAN WIJK J. J.: Squarified treemaps. In *Proc. Symposium on Visualization (VisSym'2000)* (2000), de Leeuw W. C., van Liere R., (Eds.), Eurographics Association, pp. 33–42. doi:10.1007/978-3-7091-6783-0_4. 2
- [BLC12] BAUR D., LEE B., CARPENDALE S.: Touchwave: kinetic multi-touch manipulation for hierarchical stacked graphs. In *Proc. Interactive Tabletops and Surfaces (ITS'12)* (2012), Shaer O., Shen C., Morris M. R., Horn M. S., (Eds.), ACM, pp. 255–264. doi:10.1145/2396636.2396675. 2
- [BNRB21] BOLTE F., NOURANI M., RAGAN E. D., BRUCKNER S.: Splitstreams: A visual metaphor for evolving hierarchies. *IEEE Trans. Vis. Comput. Graph.* 27, 8 (2021), 3571–3584. doi:10.1109/TVCG.2020.2973564. 3, 10
- [BW08] BYRON L., WATTENBERG M.: Stacked graphs - geometry & aesthetics. *IEEE Trans. Vis. Comput. Graph.* 14, 6 (2008), 1245–1252. doi:10.1109/TVCG.2008.166. 2
- [BWB*14] BECK F., WISZNIIEWSKY F., BURCH M., DIEHL S., WEISKOPF D.: Asymmetric visual hierarchy comparison with nested icicle plots. In *Proc. Workshop on Euler Diagrams and Graph Visualization in Practice* (2014), Burton J., Stapleton G., Klein K., (Eds.), vol. 1244 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 53–62. URL: <https://ceur-ws.org/Vol-1244/GViP-paper3.pdf>. 3
- [CBI*05] CAMARRI S., BUFFONI M., IOLLO A., SALVETTI M. V., ET AL.: Simulation of the three-dimensional flow around a square cylinder between parallel walls at moderate reynolds numbers. In *XVII Congresso AIMETA di Meccanica Teorica e Applicata, Volume II* (2005), vol. 1, Firenze University Press, pp. 23–34. 8
- [CSWP18] CUENCA E., SALLABERRY A., WANG F. Y., PONCELET P.: Multistream: A multiresolution streamgraph approach to explore hierarchical time series. *IEEE Trans. Vis. Comput. Graph.* 24, 12 (2018), 3160–3173. doi:10.1109/TVCG.2018.2796591. 2
- [dBOS13] DE BERG M., ONAK K., SIDIROPOULOS A.: Fat polygonal partitions with applications to visualization and embeddings. *J. Comput. Geom.* 4, 1 (2013), 212–239. doi:10.20382/JOCG.V4I1A. 2
- [DN23] DOBLER A., NÖLLENBURG M.: OSF: Improving temporal treemaps by minimizing crossings, 2023. doi:10.17605/OSF.IO/WZBM9. 2, 5, 10
- [EW94] EADES P., WORMALD N. C.: Edge crossings in drawings of bipartite graphs. *Algorithmica* 11, 4 (1994), 379–403. doi:10.1007/BF01187020. 3, 7
- [GG16] GEVECI B., GARTH C.: IEEEVIS: Scientific visualization contest, 2016. URL: <http://www.uni-kl.de/sciviscontest/>. 8, 10
- [GJ83] GAREY M. R., JOHNSON D. S.: Crossing number is NP-complete. *SIAM J. Algebraic Discret. Methods* 4, 3 (1983), 312–316. 5
- [GJLM16] GRONEMANN M., JÜNGER M., LIERS F., MAMBELLI F.: Crossing minimization in storyline visualization. In *Proc. Graph Drawing and Network Visualization (GD'16)* (2016), Hu Y., Nöllenburg M., (Eds.), vol. 9801 of *Lecture Notes in Computer Science*, Springer, pp. 367–381. doi:10.1007/978-3-319-50106-2_29. 3, 6
- [GJR85] GRÖTSCHHEL M., JÜNGER M., REINELT G.: Facets of the linear ordering polytope. *Math. Program.* 33, 1 (1985), 43–60. doi:10.1007/BF01582010. 6
- [GK10] GRAHAM M., KENNEDY J. B.: A survey of multiple tree visualisation. *Inf. Vis.* 9, 4 (2010), 235–252. doi:10.1057/IVS.2009.29. 3
- [GLAK23] GRAY K., LI M., AHMED R., KOBouROV S.: Visualizing evolving trees. In *Proc. Graph Drawing and Network Visualization (GD'22)* (2023), Angelini P., von Hanxleden R., (Eds.), vol. 13764 of *Lecture Notes in Computer Science*, Springer, pp. 319–335. doi:10.1007/978-3-031-22203-0_23. 1, 2
- [GPPS13] GÓMEZ J. A. G., PACK M. L., PLAISANT C., SHNEIDERMAN B.: Visualizing change over time using dynamic hierarchies: Treiversity2 and the stemview. *IEEE Trans. Vis. Comput. Graph.* 19, 12 (2013), 2566–2575. doi:10.1109/TVCG.2013.231. 3
- [GSWD18] GÖRTLER J., SCHULZ C., WEISKOPF D., DEUSSEN O.: Bubble treemaps for uncertainty visualization. *IEEE Trans. Vis. Comput. Graph.* 24, 1 (2018), 719–728. doi:10.1109/TVCG.2017.2743959. 2
- [Gur23] GUROBI OPTIMIZATION, LLC: Gurobi Optimizer Reference Manual, 2023. URL: <https://www.gurobi.com>. 8
- [HHWN02] HAVRE S., HETZLER E. G., WHITNEY P., NOWELL L. T.: Themeriver: Visualizing thematic changes in large document collections. *IEEE Trans. Vis. Comput. Graph.* 8, 1 (2002), 9–20. doi:10.1109/2945.981848. 2
- [HN14] HEALY P., NIKOLOV N. S.: Hierarchical drawing algorithms. In *Handbook of Graph Drawing and Visualization*, Tamassia R., (Ed.), CRC Press, 2014, ch. 13, pp. 409–454. 3
- [HTMD14] HAHN S., TRÜMPER J., MORITZ D., DÖLLNER J.: Visualization of varying hierarchies by stable layout of voronoi treemaps. In *Proc. International Conference on Information Visualization Theory and Applications (IVAPP'2014)* (2014), Laramée R. S., Kerren A., Braz J., (Eds.), SciTePress, pp. 50–58. doi:10.5220/0004686200500058. 2
- [JLM98] JÜNGER M., LEIPERT S., MUTZEL P.: Level planarity testing in linear time. In *Proc. Graph Drawing and Network Visualization (GD'98)* (1998), Whitesides S., (Ed.), vol. 1547 of *Lecture Notes in Computer Science*, Springer, pp. 224–237. doi:10.1007/3-540-37623-2_17. 3
- [JS91] JOHNSON B., SHNEIDERMAN B.: Tree maps: A space-filling approach to the visualization of hierarchical information structures. In *Proc. IEEE Visualization Conference (IEEE Vis'91)* (1991), Nielson G. M., Rosenblum L. J., (Eds.), IEEE Computer Society Press, pp. 284–291. doi:10.1109/VISUAL.1991.175815. 2
- [KNP*15] KOSTITSYNA I., NÖLLENBURG M., POLISHCHUK V., SCHULZ A., STRASH D.: On minimizing crossings in storyline visualizations. In *Proc. Graph Drawing and Network Visualization (GD'15)* (2015), Giacomo E. D., Lubiw A., (Eds.), vol. 9411 of *Lecture Notes in Computer Science*, Springer, pp. 192–198. doi:10.1007/978-3-319-27261-0_16. 3

- [KW19] KÖPP W., WEINKAUF T.: Temporal treemaps: Static visualization of evolving trees. *IEEE Trans. Vis. Comput. Graph.* 25, 1 (2019), 534–543. doi:10.1109/TVCG.2018.2865265. 1, 2, 3, 4, 5, 6, 8, 10
- [LWM*17] LUKASCZYK J., WEBER G. H., MACIEJEWSKI R., GARTH C., LEITTE H.: Nested tracking graphs. *Comput. Graph. Forum* 36, 3 (2017), 12–22. doi:10.1111/cgf.13164. 1, 2, 3
- [LZD*20] LI G., ZHANG Y., DONG Y., LIANG J., ZHANG J., WANG J., MCGUFFIN M. J., YUAN X.: BarcodeTree: Scalable comparison of multiple hierarchies. *IEEE Trans. Vis. Comput. Graph.* 26, 1 (2020), 1022–1032. doi:10.1109/TVCG.2019.2934535. 3
- [MSM99] MATUSZEWSKI C., SCHÖNFELD R., MOLITOR P.: Using sifting for k-layer straightline crossing minimization. In *Proc. Graph Drawing and Network Visualization (GD'99)* (1999), Kratochvíl J., (Ed.), vol. 1731 of *Lecture Notes in Computer Science*, Springer, pp. 217–224. doi:10.1007/3-540-46648-7_22. 10
- [Mun09] MUNROE R.: Movie Narrative Charts, Nov. 2009. URL: <https://xkcd.com/657/>. 3
- [Pur97] PURCHASE H. C.: Which aesthetic has the greatest effect on human understanding? In *Proc. Graph Drawing and Network Visualization (GD'97)* (1997), Battista G. D., (Ed.), vol. 1353 of *Lecture Notes in Computer Science*, Springer, pp. 248–261. doi:10.1007/3-540-63938-1_67. 2, 10
- [Sch11] SCHULZ H.-J.: Treevis.net: A tree visualization reference. *IEEE Computer Graphics and Applications* 31, 6 (2011), 11–15. doi:10.1109/MCG.2011.103. 1, 2
- [SFL10] SUD A., FISHER D., LEE H.: Fast dynamic voronoi treemaps. In *Proc. Symposium on Voronoi Diagrams in Science and Engineering (ISVD'2010)* (2010), Mostafavi M. A., (Ed.), IEEE Computer Society, pp. 85–94. doi:10.1109/ISVD.2010.16. 2
- [Shn92] SHNEIDERMAN B.: Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.* 11, 1 (1992), 92–99. doi:10.1145/102377.115768. 1, 2
- [SSV18] SONDAG M., SPECKMANN B., VERBEEK K.: Stable treemaps via local moves. *IEEE Trans. Vis. Comput. Graph.* 24, 1 (2018), 729–738. doi:10.1109/TVCG.2017.2745140. 2
- [STT81] SUGIYAMA K., TAGAWA S., TODA M.: Methods for visual understanding of hierarchical system structures. *IEEE Trans. Syst. Man Cybern.* 11, 2 (1981), 109–125. doi:10.1109/TSMC.1981.4308636. 3, 7
- [SW01] SHNEIDERMAN B., WATTENBERG M.: Ordered treemap layouts. In *Proc. IEEE Symposium on Information Visualization (InfoVis'01)* (2001), Andrews K., Roth S. F., Wong P. C., (Eds.), IEEE Computer Society, pp. 73–78. doi:10.1109/INFVIS.2001.963283. 2
- [TA08] TELEA A. C., AUBER D.: Code flows: Visualizing structural evolution of source code. *Comput. Graph. Forum* 27, 3 (2008), 831–838. doi:10.1111/J.1467-8659.2008.01214.X. 3
- [TC13] TAK S., COCKBURN A.: Enhanced spatial stability with hilbert and moore treemaps. *IEEE Trans. Vis. Comput. Graph.* 19, 1 (2013), 141–148. doi:10.1109/TVCG.2012.108. 2
- [vDFF*17] VAN DIJK T. C., FINK M., FISCHER N., LIPP F., MARKFELDER P., RAVSKY A., SURI S., WOLFF A.: Block crossings in storyline visualizations. *J. Graph Algorithms Appl.* 21, 5 (2017), 873–913. doi:10.7155/JGAA.00443. 3
- [vDLMW17] VAN DIJK T. C., LIPP F., MARKFELDER P., WOLFF A.: Computing storyline visualizations with few block crossings. In *Proc. Graph Drawing and Network Visualization (GD'17)* (2017), Frati F., Ma K., (Eds.), vol. 10692 of *Lecture Notes in Computer Science*, Springer, pp. 365–378. doi:10.1007/978-3-319-73915-1_29. 3
- [vFWTS08] VON FUNCK W., WEINKAUF T., THEISEL H., SEIDEL H.: Smoke surfaces: An interactive flow visualization technique inspired by real-world flow experiments. *IEEE Trans. Vis. Comput. Graph.* 14, 6 (2008), 1396–1403. doi:10.1109/TVCG.2008.163. 8
- [vHH17] VAN HEES R., HAGE J.: Stable and predictable voronoi treemaps for software quality monitoring. *Inf. Softw. Technol.* 87 (2017), 242–258. doi:10.1016/J.INFSOF.2016.10.003. 2
- [VSC*20] VERNIER E. F., SONDAG M., COMBA J. L. D., SPECKMANN B., TELEA A. C., VERBEEK K.: Quantitative comparison of time-dependent treemaps. *Comput. Graph. Forum* 39, 3 (2020), 393–404. doi:10.1111/CGF.13989. 2
- [vWvdW99] VAN WIJK J. J., VAN DE WETERING H.: Cushion treemaps: Visualization of hierarchical information. In *Proc. IEEE Symposium on Information Visualization (InfoVis'99)* (1999), IEEE Computer Society, pp. 73–78. doi:10.1109/INFVIS.1999.801860. 2
- [WK06] WATTENBERG M., KRISS J.: Designing for social data analysis. *IEEE Trans. Vis. Comput. Graph.* 12, 4 (2006), 549–557. doi:10.1109/TVCG.2006.65. 2
- [WPCM02] WARE C., PURCHASE H. C., COLPOYS L., MCGILL M.: Cognitive measurements of graph aesthetics. *Inf. Vis.* 1, 2 (2002), 103–110. doi:10.1057/PALGRAVE.IVS.9500013. 2, 10