UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Algorithmic Drawing of Evolving Trees

Algorithmisches Zeichnen zeitlich veränderlicher Baumstrukturen

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

Vorgelegt von
Malte Skambath

Ausgegeben und betreut von
Prof. Dr. Till Tantau

Lübeck, den 7. Mai 2016

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, den 7. Mai 2016

Abstract

In computer science, many data structures can be represented as *trees* and their visualization has become an important and broadly studied task in the field of algorithmic graph drawing. Since data structures often change over time, whole sequences of related trees — called *evolving* trees — often need to be visualized as whole, but existent approaches process trees in such a sequence incrementally or obscure the whole nature of the changes. In this thesis we consider the algorithmic generation of animations for evolving trees as an offline problem. We identify objectives for such animations and discuss the complexity and feasibility of meeting them. This leads to a new NP-complete problem. A new algorithm with a heuristic approach for arbitrary evolving trees is developed, and its implementation is presented as a proof-of-concept for animated graph drawing in the TEX framework Ti*k*Z. The algorithm is applied to several real-world examples and the resulting animations demonstrate that it works well in these cases.

Zusammenfassung

In der Informatik gibt es viele Datenstrukturen in Form von *Bäumen*, deren Darstellung im Bereich des algorithmischen Graphzeichnens eine wichtige und weit erforschte Aufgabe geworden ist. Da sich Datenstrukturen häufig verändern, müssen oft ganze Folgen von Bäumen dargestellt werden. Existierende Ansätze erzeugen Darstellungen von Bäumen meist inkrementell oder verbergen charakteristische Eigenschaften der Änderungen. In dieser Arbeit wird die algorithmische Erzeugung von Animationen zeitlich verändernder Bäume als Offline-Problem betrachtet. Hierbei werden Kriterien für die Darstellung und Animation erarbeitet und die Machbarkeit diese umzusetzen untersucht. Dabei zeigt sich ein neues NP-vollständiges Problem. Ein neuer Algorithmus mit heuristischem Ansatz für beliebige zeitlich verändernde Bäume wird entwickelt und seine Implementierung als Proof-of-Concept für animiertes Graphzeichnen mithilfe des TEX Frameworks Ti*k*Z vorgestellt. Der Algorithmus wird auf verschiedene Praxisbeispiele angewendet und die erzeugten Animation zeigen, dass er hierbei gute Ergebnisse erzielt.

This thesis was written entirely in LaTeX and exists in two versions: one paper or PDF version and one animated version in the SVG-format. If the first image in a sequence of images has a filled background, then a click on this background starts an animation of the sequence.

Contents

1 Introduction

Trees are well known mathematical structures that can be found in different areas. In computer science, trees are used in many data structures like prefix trees, parse trees, or binary search trees. Since visualization helps understanding data represented by trees, the automatic generation of visual pleasing and useful representations is an important and broadly studied task. Usually trees are visualized as hierarchical structures and drawn downwards starting from the root node. However, data structures often change over time. For example, by the insertion or deletion of nodes in a search tree as seen in Figure 1.1. Drawing the evolution of such a tree leads to new problems and possibilities: We can visualize temporal information by using multiple drawings or animations, but we have to take care that a viewer can follow the changes.

A naive approach to visualize such an *evolving* tree uses an existent algorithm for *static* trees and places the resulting drawings next to each other. This allows a viewer to understand each single drawing and to recognize the changes in the tree. For this, the drawings have to be compared pairwise to see the changes. If the positions of some nodes change between drawings, then following them might be more difficult for a viewer, but it could also help in some cases to get the right idea of what happens. For example, there is a difference in seeing that certain edges appear and disappear in comparison to the perception that a subtree has been pruned and regrafted at another position in the tree. Moreover, a sequence of drawings may not be the preferable choice. Since the time dimension allows the use of animations, these can be used to improve the abilities of viewers to follow nodes and to see the relevant changes without the need to compare consecutive drawings. If nodes or edges change their positions, the changes can be visualized by smooth transitions between two drawings.

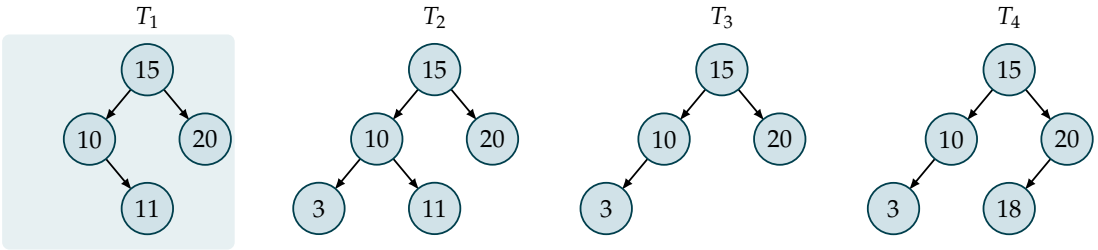


Figure 1.1: A binary search tree that changes over time.

Using animations with a naive algorithm that creates one drawing for each state of the tree independently is still not the best approach. It might happen that multiple nodes move simultaneously or with a high speed, which makes it difficult to follow them, even with smooth transitions. While drawings must be readable and consistent, viewers need a chance to easily follow changes such that they get the right and intended understanding of what they see. Therefore, we need some criteria or objectives that try to preserve the *mental map* of a viewer.

1.1 Main Contributions

When evolving trees are used in presentations, for example in computer science lectures about search trees, their structure is fixed before they are shown. While existent algorithms for evolving trees are online approaches that create animations by producing drawings or layouts incrementally, thereby causing unnecessary movements of nodes, it is desirable to have animations that profit from the setting that evolving trees are known as whole before they are visualized. In this thesis we investigate the visualization of evolving trees as such an offline problem.

We review known criteria for static drawings of trees and elaborate new criteria, which preserve the mental map in animations such that drawings of consecutive states of the graph are similar and a viewer can easily follow the changes. I developed a new algorithm for evolving trees that preserves the specified criteria as well as possible. It reduces unnecessary movements in an animation while it tries to save space. I was able to show that violations of the specified criteria cannot be avoided completely. Furthermore, the minimization of those violations turns out to be an NP-complete problem, which is the reason why I propose a heuristic. The algorithm allows an arbitrary set of updates or changes in contrast to other algorithms.

The whole algorithm was implemented in the Lua programming language for the Ti*k*Z framework. Ti*k*Z is an extensive graphics library for the typesetting system TEX and was developed by Till Tantau [41]. In a recent version, Ti*k*Z provides the possibility to create animations in the scalable vector graphics format SVG. The implementation of the new algorithm is a first prototype for drawing evolving graphs in Ti*k*Z. Using this prototype, the example shown in Figure 1.1 can be produced with the following Ti*k*Z-code:

```
\tikz\graph[animated binary tree layout, ...]
{
  {[when=1] 15 -> {10 -> { ,11},  20      }}, % T_1
  {[when=2] 15 -> {10 -> {3,11},  20      }}, % T_2
  {[when=3] 15 -> {10 -> {3,  },  20      }}, % T_3
  {[when=4] 15 -> {10 -> {3,  },  20 ->18 }}, % T_4
};
```

Listing 1.1: A first example of an evolving tree in Ti*k*Z.

1.2 Related Work

The algorithmic generation of hierarchical drawings of binary trees has been explored intensively in the past. First graph drawing approaches for trees were developed by Knuth, Wetherell and Shannon, and Sweet [27, 39, 43]. Those early approaches had different drawbacks. For example, Knuth's approach may cause unnecessarily wide drawings. To achieve "tidy" drawings of trees, Wetherell and Shannon identified specific criteria, which seem to be reasonable in a drawing. They provided an algorithm that tries to respect these aesthetic criteria and achieve drawings with a small width. Still, their algorithms produce drawings that do not have optimal width.

Reingold and Tilford [34] brought up a new aesthetic criterion to achieve better results. They suggested that symmetric tree structures should be drawn symmetrically and provided an algorithm which supports this objective well and runs in linear time. Later the algorithm was improved for unbounded-ary trees by Walker [42] and it was claimed and finally shown that this extension can be computed in linear time, too [8, 42]. While the specified criteria describe the structure of a drawing explicitly, it was always an objective to have narrow drawings. It was shown that the drawing of binary trees of minimum width and with respect to the specified criteria can be solved by a linear program in polynomial time, but turned out to be an NP-complete problem for the discrete case with integral coordinates. This was shown by reducing 3-SAT on the drawing problem for a fixed minimum width [38].

Other algorithms often use Reingold and Tilford's approach. Brüggemann and Wood [6] implemented it with a few improvements entirely in the typesetting system TEX. Later Tantau reimplemented the algorithm with these improvements for the graph drawing engine in Ti*k*Z [40, 41].

Trees do not have to be visualized by node-link diagrams. There are also other techniques like tree maps [25], three dimensional cone trees [35], or sunburst visualizations [36]. For evolving graphs, pretty much as for static graphs, there are several techniques for visualizations, too. Straightforwardly, they can be visualized by a sequence of static drawings or by animations that smoothly visualize the evolution of a these drawings [17]. It is possible to adapt this concept and use the time as another space dimension. Then nodes are drawn as tubes through the space [23]. There are different techniques not restricted to node-link diagrams [9, 10, 22, 33]. For instance, matrix cubes [3], as a three dimensional extension of adjacency matrices. One layer in such cube represents the adjacency matrix in one state of the graph. An extensive overview of the whole state of the art including a taxonomy of different visualization techniques is given by Beck et al. in [4]. Since node-link diagrams are intuitive and predominant, we focus on animated node-link diagrams in this thesis.

Many approaches in drawing evolving trees create drawings incrementally and expect a sequence of update operation causing the changes [12, 30]. Those algorithm are designed for interactive software and create or adjust the layout for each change. This implies that each drawing only depends on the change and previous drawings.

An early algorithm designed for evolving trees was developed by Moen [30]. It is based on Reingold and Tilford's static algorithm and places nodes and subtrees close together by using the shape or user-defined borderlines of nodes. Instead of recomputing the whole layout on each change, Moen's algorithm provides a specific set of update operations that adjust the current layout. Later Cohen et al. [11, 12] presented algorithms for different families of graphs including trees. Their algorithm also provides a fixed set of update operations. The layouts produced with their algorithm were wider than those of Moen, but each update operation can be performed in logarithmic time.

Diehl, Görg and Kerren [15, 16] introduced a general concept, called *foresighted layout*, for drawing evolving graphs offline. They introduced the *supergraph*, as the graph we achieve by merging all states of an evolving graph, and they proposed to use a static graph drawing algorithm on this graph to compute fixed positions for the nodes. As this results in drawings of unnecessary big size, they suggested to reduce number of nodes by using the fact that nodes that do not exist at the same time may share the same position and can be seen as one single node. Reducing the number of nodes was shown to be an NP-complete problem.

Later they generalized their idea and allowed a few number of nodes to change their position [14]. This extension computes one static layout for each state of the graph and the supergraph respectively the reduced supergraph. Finally, all those static layouts are used to produce a final sequence of layouts by applying a proposed adjustment strategy. This approach was used on evolving hierarchical graphs [21].

1.3 Structure of this Thesis

In this thesis we focus on the visualization of evolving trees and how to compute well layouts such that they can be animated. In Chapter 2, I introduce the terminology for evolving trees used in this thesis and present the problem of drawing evolving graphs.

In Chapter 3, general offline approaches will be presented and we identify important criteria for animations and the weaknesses of these general approaches in the context of evolving trees.

In Chapter 4, I present criteria for static and evolving trees. We review the algorithm by Reingold and Tilford for the static case and develop a new algorithm for arbitrary evolving trees that reduces unnecessary motions with a heuristic approach. Finally, I prove that meeting desired criteria is not possible in some cases and show that the minimization of such violations turns out to be a new NP-complete problem.

Chapter 5 presents the use of graph drawing in Ti*k*Z, introduces the new extension to evolving graphs, and describes relevant concepts of the prototype implementation of the previously presented algorithms.

Finally, in Chapter 6 we sum up the results of this thesis and suggest where the supplied work and research on the algorithms, the prototype, or the conceptual view could be continued.

2 Background: Evolving Graphs and Their Layouts

Graphs, including trees, can be found in an enormous number of applications. They consist of nodes and edges connecting these nodes. Graphs are understandable in a natural way and their simple concept allows several modification or extensions: For example, there are directed, undirected graphs, multi- or hypergraphs. In addition, graphs are usually more than just nodes and edges. Nodes and edges may have labellings, colors, weights, or other attributes. As this holds also for evolving graphs, their exact definition is not a trivial problem.

There are different formal definitions of evolving graphs. In my view, these can be divided into two basic ideas: Firstly, an update based model describes an evolving graph by a sequence of single *update operations* that consecutively change an initial graph. Such an update operation describes the changes between two states of the graph and could be a set of appearing and disappearing nodes and edges [31]. An advantage of this model is its intuitiveness in the context of algorithms. In search trees we have update operations like the insertion or deletion of single nodes. In practice, the set of updates is usually limited and does not allow arbitrary changes [11, 12, 30]. Secondly, a state based model represents an evolving graph directly as a sequence of single graphs [15]. Each of them is just one state of the graph. They can differ in their sets of nodes and edges but may have some common items too.

The update and state based models are equivalent. Given a sequence of update operations, we can simulate these updates incrementally on the initial graph to get a sequence of graphs. The construction in the other direction may be more difficult. If arbitrary changes are allowed, then the set of nodes and edges between consecutive graphs can be compared. Otherwise, further analysis might be necessary.

Since update operations can be transformed into graph sequences and since the algorithms in the following chapters are independent of a restricted set of them, our definition for evolving graphs will be state based. In this chapter we define evolving graphs and useful terms for formalization of related problems. A short introduction into the basic problems of graph drawing and and animated graph drawing for evolving graphs and trees is given.

2.1 Terminology for Evolving Graphs and Trees

Depending on the context, exact graph models differ. In this thesis, every *graph* $G = (V, E)$ consists of *nodes* or *vertices* V and *edges* E connecting pairs of nodes. In *directed graphs* we define edges as ordered pairs of nodes $E \subseteq V \times V$, while in an *undirected graph* the edges are defined as two element subsets of nodes $E \subseteq \{\{v, w\} \mid v, w \in V, v \neq w\}$. In an *annotated graph* $G = (V, E, \varphi)$ we can describe properties like shapes, weights, colorings, or labels using a function $\varphi : V \cup E \to A$, where A is a set of possible attribute vectors. For general considerations, we assume to have annotated graphs but we do not fix the edge model unless it is required or not clear by the context. *Trees* are acyclic and connected graphs.

In real world applications, graphs can change their structure. We call such a graph that "evolves" over time an *evolving graph*. The terms *temporal* [28,44] or *dynamic graph* [11,20] are commonly used too. For better distinctions we denote a "normal" graph as *static graph*.

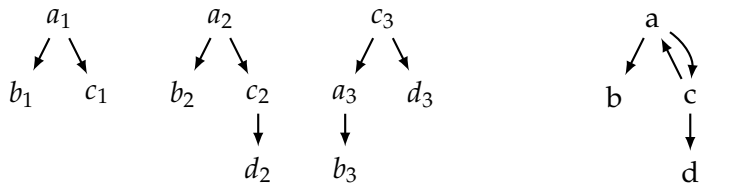
Definition 2.1 (Evolving Graph)**.** *An* evolving graph $\mathcal{G} = (G_1, \ldots, G_n)$ *is a sequence of graphs* $G_i = (V_i, E_i, \varphi_i)$.

For a given evolving graph $\mathcal{G} = (G_1, \ldots, G_n)$ we call a single graph in the sequence a *snapshot of* $\mathcal{G}$ and write $|\mathcal{G}|$ to denote the number of snapshots in $\mathcal{G}$. As we deal with trees in this thesis we define an *evolving tree* naturally as an evolving graph where each snapshot is a tree:

Definition 2.2 (Evolving Tree)**.** *An* evolving tree *is an evolving graph* $\mathcal{T}$ *where each snapshot* T_i *is a tree.*

The *supergraph* [16] is a merge of all snapshots and helps analyzing evolving graphs and will be used later in the algorithms:

Definition 2.3 (Supergraph)**.** *Let* $\mathcal{G} = (G_1, \ldots, G_n)$ *be an evolving graph. The* supergraph of $\mathcal{G}$ *is a graph* $\widehat{\mathcal{G}} = (\widehat{V}, \widehat{E})$ *with* $\widehat{V} := \bigcup_{i=1}^{n} V_i$ *and* $\widehat{E} := \bigcup_{1}^{n} E_i$ *as the set of nodes and edges that exists in at least one snapshot of* $\mathcal{G}$.



(a) Snapshots **(b)** Supergraph

Figure 2.1: An evolving graph with its sequence of graphs (a) and the supergraph (b). The labelled nodes a_i represents the same node a just in a different snapshot.

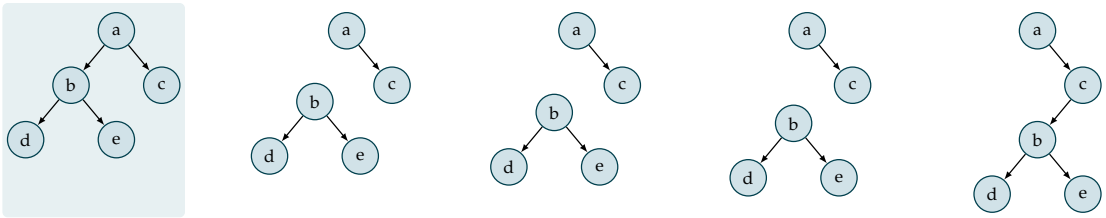


Figure 2.2: An animation provides a smooth transition between the drawings of two consecutive snapshots.

Note that a condition that holds for every snapshot must not necessarily hold on the supergraph and vice versa. Given an evolving tree $\mathcal{T}$ like the one shown in Figure 2.1, the directed supergraph $\widehat{\mathcal{T}}$ can be cyclic or a directed acyclic graph that is no tree.

2.2 Drawing Graphs and Evolving Graphs

Graph drawing addresses the problem of visualizing graphs. There are different visualization techniques for graphs or trees with certain advantages. For example, treemaps [25] or node-link diagrams. However, node-link diagrams are intuitive and predominant. In a *node-link diagram*, nodes are naturally drawn as dots or in certain shapes and edges are represented by geometrical lines connecting the nodes. Those lines can be straight lines but also polylines or Bézier curves.

In data visualization, attributes like color, text labels, and shapes can be applied directly to a drawing. The problem of graph drawing can be structured in several phases like the node positioning and edge routing. Since trees are mostly drawn with straight lines, we only focus on the placement of nodes and the straight lines of the edges are implied by the positions of nodes. The shape and size of nodes may be important for the placement as nodes should not overlap in a drawing.

Given a graph $G = (V, E)$, a *layout* L of G is a function $V \to \mathbb{R}^2$ mapping each node $v \in V$ to a coordinate in the two-dimensional plane. With $L_x(v)$ and $L_y(v)$ we denote the x and y-coordinate of $L(v)$. Remember that our goal is to find an animation that smoothly visualize an evolving graph. Therefore, each snapshot has to be drawn once and requires an own layout:

Definition 2.4 (Evolving Graph Layout)**.** *Let $\mathcal{G}$ be an evolving graph. An* evolving graph layout *for $\mathcal{G}$ is a sequence of graph layouts $\mathcal{L} = (L_1, \ldots, L_{|\mathcal{G}|})$ such that L_i is a layout of G_i for all $i \in \{1, \ldots, |\mathcal{G}|\}$.*

An animation extends a discrete evolving graph layout as a continuous progress of layouts. To get smooth animations, layouts between two snapshots linear interpolation can be used as shown in Figure 2.2. It is desirable that the duration between consecutive snapshots is not constant. We use a mapping between the discrete steps

$\{1, \ldots, |\mathcal{G}|\}$ and times in $\mathbb{R}$ to specify the time when the snapshots are shown in an animation. Such a time mapping has to respect the ordering of the snapshots in the evolving graph:

Definition 2.5 (Time Mapping Function)**.** *Given an evolving graph $\mathcal{G}$, a* time mapping *is a strictly monotone increasing function $\tau : \{1, \ldots, |\mathcal{G}|\} \to \mathbb{R}$.*

In static graph drawing problems, algorithms try to preserve specified objectives. For example, topological hierarchies of graphs should be visualized by the vertical ordering of nodes, nodes must have a minimal distance, or the drawing as whole does not exceed a specified size or aspect ratio. Some criteria are defined for specific graph families. It is reasonable to have the same or similar criteria for the snapshot layouts, but this alone is not enough for animations. Since an animation has to mediate certain coherences between consecutive drawings to preserve the mental map of a viewer [16, 29], further criteria that guarantee this *stability* [32] over the temporal evolution are necessary.

Cohen et al. [11, 12] divided objectives or criteria for evolving graph layouts into two types: the *static drawing predicates* preserving the readability in the layouts of the snapshots and the *dynamic drawing predicates* for objectives on the temporal evolution preserving the stability. In the following we call these predicates the *static* and *dynamic aesthetic criteria*. Graph drawing for evolving graphs addresses the problem of finding an evolving graph layout meeting desired criteria. Since most sets of criteria are contradictory, some trade-offs or priorities might be necessary.

In animations, moving nodes have speeds an also influence how good a viewer can follow some changes and whether an animation preserves the mental map. There are some factors that influence the speed. The distance of the node positions in consecutive layouts for the same node and the shape of the motion paths. In this thesis we assume that nodes move on straight lines with a constant speed, but we should keep in mind that interpolation is not the best solution [18]. With that assumption, the quality of an animations depends on the time mapping and the distances in the layout. For trees, the measurement of speed will be less important since time mappings are provided by users and in the best case nodes only move on pruning subtrees. In this case, the structure of the tree is more important than the speed of moving a subtree.

3 Methodology: Approaches in Drawing Evolving Graphs

There are several approaches in drawing evolving graphs and trees. Most of them are online approaches. If we visualize an evolving graph in a document or a presentation, the whole graph is available for the typesetting and drawing program. In such settings offline algorithms seem to be the first choice.

In this chapter we discuss the differences between on- and offline drawing problems. We review offline approaches that adapt layout algorithms for static graphs. Those algorithms for arbitrary evolving graphs promise more stable layouts than online approaches by suppressing changes of node positions. We identify the issues of these general approaches and recognize that, especially for trees, layouts without any movement of nodes are unpractical and difficult to understand.

3.1 Online versus Offline Approaches

Online algorithms for drawing evolving graphs generate a new layout for each snapshot without having access to subsequent snapshots. This is the setting in interactive graph drawing or visualization systems where a user directly or indirectly modifies a graph and want to see the new layout or the transformation into a new one instantly.

While the future is unknown, it is impossible to predict possible changes and to avoid dissimilarities between consecutive layouts. Sometimes, it is necessary to move nodes when space for a new unpredictable node is needed. The challenge in the online problems is to guarantee readability while structural similarities, for example by the vertical and horizontal ordering of nodes, in consecutive drawing should be preserved.

Ordered trees like binary or n-ary trees induce an own ordering and a hierarchy of nodes. For that reason suitable layouts for static trees can be used for each single snapshot to preserve structural similarities. The insertion of a node in a tree may result in shifted node positions but the ordering and hierarchy are preserved automatically by the given tree. Note that this is not that simple for other graph families like directed acyclic graphs where no ordering in the horizontal direction exists [32]. Although can produce acceptable layouts, in some cases it is still difficult to preserve the mental

map as unnecessary movements of nodes can happen.

An offline algorithm has access to all snapshots before it computes a whole layout for an evolving graph. Offline algorithm can simulate online algorithms such that we can expect that the layouts are at least as good as those generated by an online algorithm. One can expect that some undesirable effects like unnecessary movements of nodes in a tree are avoidable since the algorithm has the ability to predict that more space is necessary for appearing nodes in some snapshots.

The use of animated graph drawing for documents or presentations perfectly fits with the conditions of an offline problem. It would be wasteful not to profit from the advantages of an offline problem. However, there seem to be no offline algorithm for trees and we see in the next sections that general approaches are not suitable and even worse than existent online approaches.

3.2 A Generic Reduction to the Static Case

Diehl, Görg and Kerren [15, 16] studied and formalized the offline problem for general and directed acyclic graphs. They introduced the concept of *foresighted layout* as a generic reduction to the static layout algorithms. It was proposed to run a reasonable layout algorithm on the supergraph and use the computed node positions in all snapshot layouts.

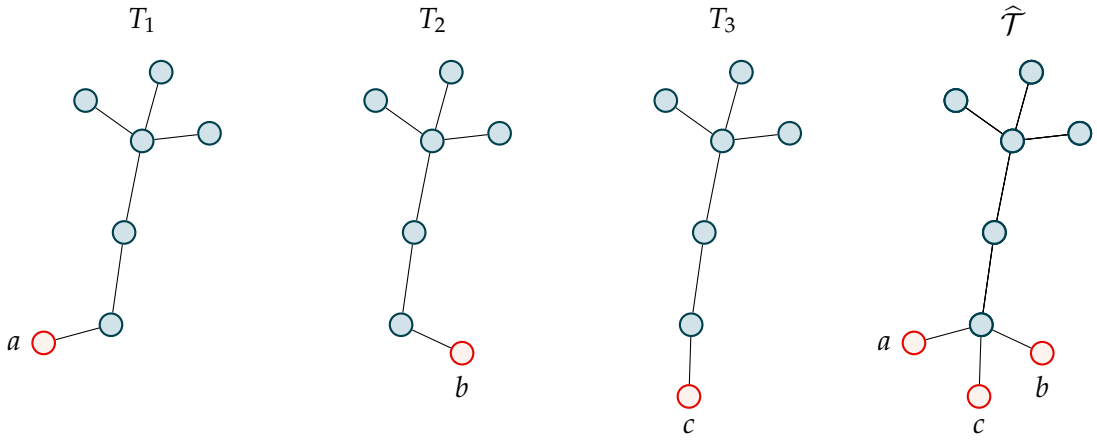


Figure 3.1: An unordered evolving tree and its supergraph $\widehat{T}$ drawn with the foresight layout. The nodes a, b, and c are drawn at different positions because they are different nodes and a static algorithms prevents nodes to have the same position.

It turned out that using a layout for the supergraph may result in drawings with unnecessary big size. Nodes which do not exist at the same time are placed onto different positions, as shown in Figure 3.1. Diehl et al. already recognized this issue and proposed to reduce the supergraph with a partitioning of nodes. A partition may contain nodes, that have no common snapshots in which they exist, such that all

nodes of a partition can share the same position and the partition can be seen as one node. Reducing the number of partitions should result in smaller drawings and in our previous example the nodes a, b, and c would be drawn at the same position.

One class of well-known static algorithms for arbitrary graphs are force-directed algorithms. They are not designed for specific families of graphs but work well on trees unless the hierarchy is important. In force-directed algorithms, nodes are simulated as physical objects with forces between them. There are repulsive force moving nodes away from each other while edges are often simulated as springs that converge to a unit length and pull connected nodes together. There are different possibilities how such forces can be defined. In the physical world systems usually approximate to stable states or states of low energy. The analogy works well for graph drawing although objectives like planarity cannot be guaranteed. Figure 3.2 shows static trees drawn with the force-directed algorithm by Fruchterman and Reingold [19].

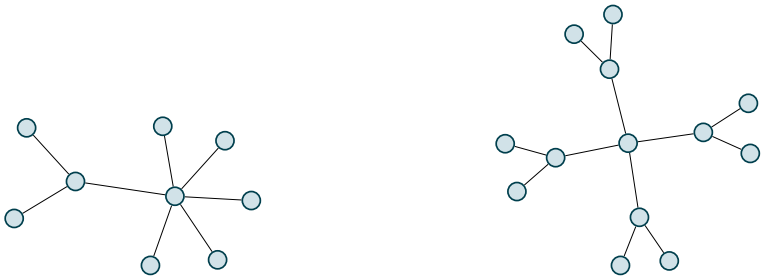


Figure 3.2: Trees that are drawn with the Fruchterman and Reingold force-directed algorithm.

However, without prior knowledge the proposed partitioning does not seem to be suitable for evolving trees. When nodes that are embedded at completely different positions in the tree structure a partitioning can result in unfitted layouts as Figure 3.3 shows.

In [16] the foresighted method and the partitioning was introduced with a tool, named GaniFA, that visualizes the construction of a nondeterministic automaton from a given regular expression. As static algorithm, they used an algorithm based on the

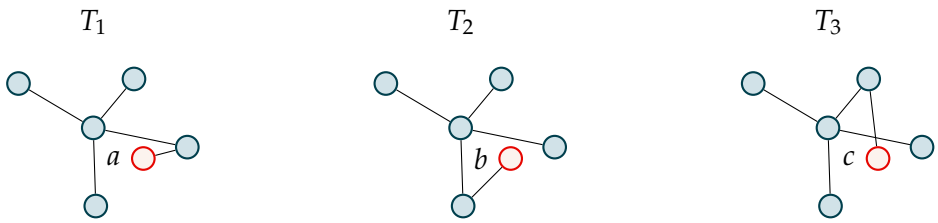


Figure 3.3: An evolving tree drawn with the foresight layout method and a force-directed algorithm. The different nodes a, b and c are drawn at the same position because they are assigned into the same partition.

Sugiyama approach for layered drawings [37]. For their purpose, the algorithm created well layouts, but unfortunately the given examples were less meaningful related to their algorithmic ideas. In the construction of an NFA with snapshot only new nodes (states) appeared. Hence, there is no node partitioning.

Although trees in our context are hierarchical structures, there is no good reason, why a static algorithm for layered graphs should be used. The foresighted approach permits any movement of nodes, such that it is easy to follow them. However, there are problems on the readability that seem to be more important than the stability: Firstly, the hierarchy might change and this cannot be visualized well by foresighted layout. Secondly, a layered layout of the supergraph does not guarantee that the snapshots of trees are drawn as planar graphs. Thirdly, in binary trees we have nodes that should be placed left and right below their parent nodes, which is not preserved by layered layout.

In further publications [14], the foresighted method was improved to the *foresighted layout with tolerance*. In this approach, a layout for all snapshot and the supergraph will be computed first. Finally, with an adjustment strategy, these layouts are used and adjusted such that a difference metric is below a specified tolerance value. This allows nodes to change their positions and preserve desired criteria. There might be a reasonable adjustment strategy for evolving trees, but we will see later in this thesis, that such an adjustment strategy is not necessary.

3.3 A Force-Directed Approach: Making Forces Time-Dependent

A problem of the approaches by Diehl et al. is that drawings either become an unnecessary size or appear in an unexpected shape by partitioning. It seems that a general partitioning is unpractical.

However, it is possible to avoid the unnecessary size of a layout without a partitioning but using a force-directed algorithm on the supergraph. We can extend a static force-directed algorithm by consideration of the temporal properties of an evolving graph. For nodes that never exist at the same time may share same position: This means there is no reason why a repulsive should move them apart from each other in the supergraph.

A time-dependent force-directed algorithm applies forces between nodes if and only if they have at least one common snapshot. When there are no forces between nodes that never exist at the same time it might happen that they are placed at the same or similar positions but only when this is forced by the structure of the evolving tree, instead of an imposed partitioning. In Figure 3.4 we can see that this algorithm avoids the unexpected effects that occurred in Figure 3.3 by a partitioning in the foresighted layout approach. However, if one nodes is connected with defferent ones some issues of the foresighted layout approach remain.

The foresighted layout approach, using a static force-directed algorithm, and the

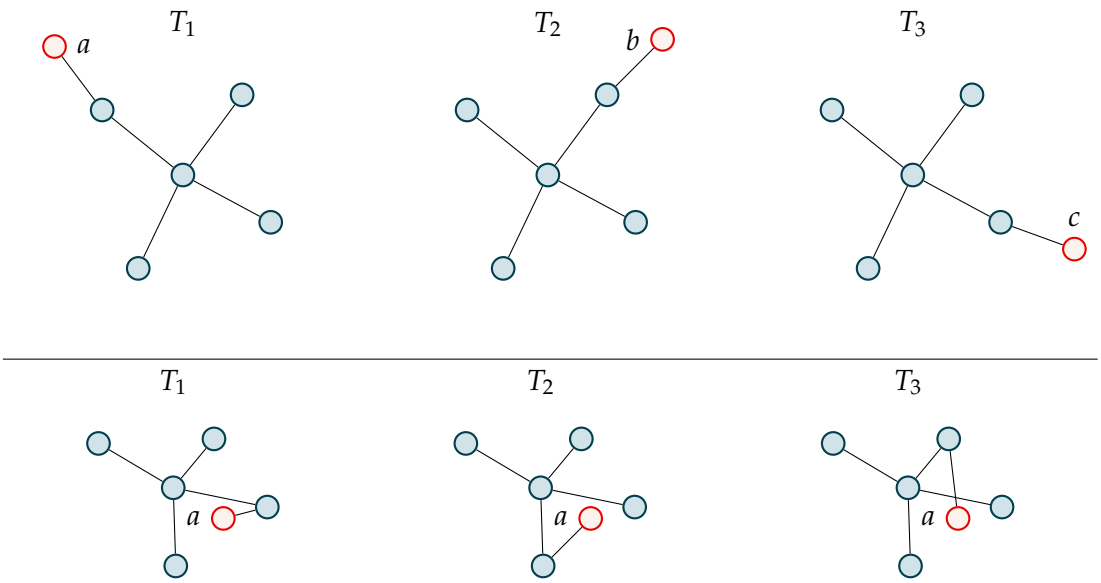


Figure 3.4: The layout of an evolving tree created by applying a force-directed algorithm on the supergraph ignoring forces between nodes that never exist at the same time. When the same node changes its connections it stays at the same place and we get undesirable results as the approach by Diehl et al.

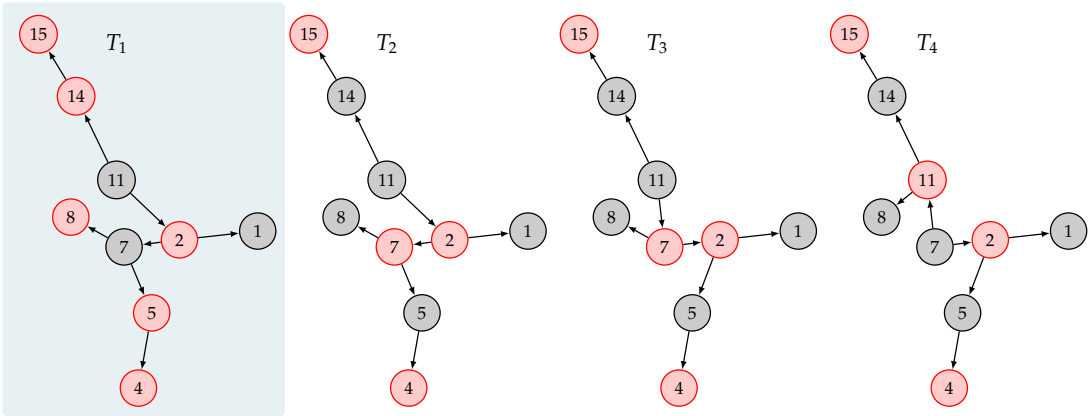


Figure 3.5: A red-black tree example taken from [13, p. 317] and visualized with a foresighted force based layout. It is difficult to recognize the changes and that subtrees has been pruned and regrafted.

improved force-directed algorithm have been implemented and tested. Unfortunately none of them convinces as a practical approach for arbitrary trees. Since motions of nodes are permitted, a viewer can easily see what happens, but might not get the intended understanding of the changes. In Figure 3.5 a change in a special search tree is shown and it is difficult to recognize that in this evolving tree some subtrees have been pruned and regrafted.

3.4 An Approach Tailored to Trees

The algorithms above were designed with priority on the stability while other important criteria on the structures of trees are ignored. This might be the reason why the approaches are not convincing to be practical for evolving trees. To get a more reasonable algorithm, we need to identify necessary criteria for evolving trees like the hierarchical structure in snapshot layout. For static trees Reingold and Tilford [34] preserve some criteria with their algorithm. This algorithm builds up a layout for a tree from the leafs up to the root and at each inner node the subtrees are placed close together.

The idea of Reingold and Tilford can be applied on evolving trees, too. We can reinterpret nodes and subtrees as three-dimensional objects with a depth that depends on the snapshot in which the nodes exist. Placing subtrees close together correspond to put three dimensional objects next to each other. Since the supergraph is not always a tree, it is not that simple. In the next chapter we take a closer look on this idea and investigate how we can support arbitrary evolving trees with an offline approach.

4 Algorithmics: Drawing Evolving Trees

Data structures like sorting trees or parsing trees are usually hierarchical and ordered. In this chapter we discuss desired criteria for evolving trees and develop a new algorithmic approach for drawing them offline.

Firstly, we review criteria for static tree layouts and identify new dynamic criteria that seem to be reasonable for animations of evolving trees. Secondly, we review Reingold and Tilford's algorithm as base for a new algorithm. Thirdly, we develop the new algorithm in three steps. We start with the strong assumption that the supergraph is a tree such that the Reingold and Tilford's algorithm can be modified for this case. In the following sections further adaptions are presented until arbitrary evolving trees are supported by the algorithm. We will see that violations of the previously identified criteria are sometimes not avoidable and a heuristic is used to reduce the number of possible violations. We conclude by proving that the problem which the heuristic tries to solve turns out to be an NP-complete problem.

4.1 Layout Objectives for Evolving Trees

In many applications we use trees to represent hierarchical and ordered data. Some special trees of that kind are binary trees. Wetherell and Shannon were one of the first who began to define aesthetics for the layout of such trees [43]. Later Reingold and Tilford [34] improved the layouts with the additional criterion that drawings of trees

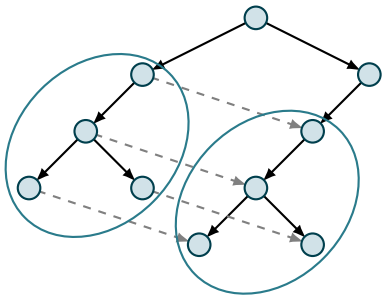


Figure 4.1: Trees are mostly drawn hierarchical with nodes centered above their children and according to Reingold and Tilford isomorphic or symmetric subtrees should be drawn similiar.

should be more symmetric and that similar subtrees achieve similar layouts. In this case, layouts of two isomorphic trees or subtrees are similar if the layout of one tree can be achieved by adding a constant vector to all node positions in the other layout, as Figure 4.1 shows. Reingold and Tilford also designed an algorithm which fulfills these aesthetic criteria.

Static Aesthetic 1 (Hierarchical Drawing)**.** *All nodes v in the same level of the tree lie on the same horizontal line. For all levels those lines should be parallel and the ordering of those nodes on the line is the same order in which the nodes appear in a traversal order of a tree.*

Static Aesthetic 2 (Left and Right Child-Placement)**.** *If a node v_ℓ is a left child, then it has to be positioned left to its parent node v and otherwise if it is a right child v_r right of its parent: $L_x(v_\ell) < L_x(v) < L_x(v_r)$.*

Static Aesthetic 3 (Parent Centering)**.** *A parent node v should be centered above its child nodes while it has at least two children v_ℓ, v_r: $L_x(v) = \frac{1}{2} \cdot (L_x(v_\ell) + L_x(v_r))$.*

Static Aesthetic 4 (Symmetrical and Isomorphic Drawings)**.** *The drawing of trees and subtrees should be symmetrical. This means that layout of a mirrored graph (every left child is a right one and vice versa) is the mirrored layout of the original graph.*

These objectives for binary trees can be adapted for non-binary trees. In this case we exclude the second criterion (SA2) and just preserve the ordering of child nodes. Thus, a single child gets the same x-coordinate as its parent node. Note that narrow images are always desirable but there is no criterion on the width or size of a drawing. Except for optimality, we cannot say that a layout is narrow or not. Width seem to be less important than the other objectives, which emphasize characteristic properties, and thus it is mostly preserved heuristically.

When we draw evolving trees we could use the same aesthetic criteria and use the algorithm of Reingold and Tilford for each single layout. Indeed, every single snapshot will be assigned to an acceptable layout. But looking at the two examples shown in Figure 4.2 we recognize that this naive approach results in many avoidable movements of nodes.

Dynamic Aesthetic 1 (Relative Node Stability)**.** *If a node w is a k-th child in any snapshot T_i, then in all snapshots T_j, where w is also the k-child of v the vector between $L_j(v)$ and $L_j(w)$ is always the same.*

The criterion for relative node stability (DA1) reduces the motion of nodes in Figure 4.2 but it does not prevent motions in general. It allows that subtrees whole subtrees can move whole their inner layout is unchanged. Although this improvement seems to be sufficient for evolving trees, we still need a new criterion which is a modified version of the criterion for symmetry in the static case (SA4).

Dynamic Aesthetic 2 (Temporal Isomorphic and Symmetric Layouts)**.** *Trees and subtrees with the same or symmetrical evolution over time get the same or symmetric inner layouts except of an offset in the position and time.*

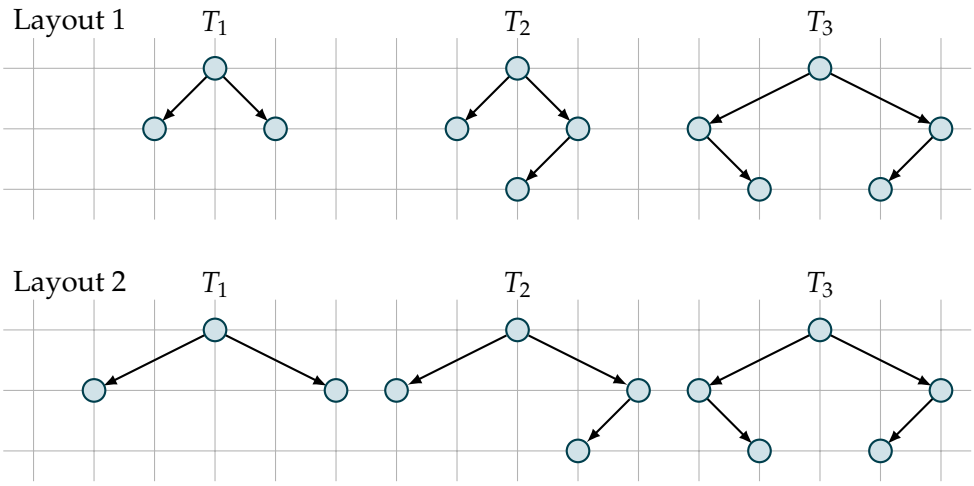


Figure 4.2: Evolving trees with a new layout for each snapshot (Layout 1) and a layout taking the whole progress of the tree into consideration (Layout 2). In Layout 1 the children of the root change their position in T_3 although this can be avoided.

The criterion (DA2) is implied by (SA4), but it is needed as a weaker replacement since (SA4) and (DA1) together are too strong. In Figure 4.3 we have a layout that fails criterion (SA4) in the first snapshot. It can be fixed by increasing the distances in the left subtree. However, this produce unnecessary wide layouts and there are evolving trees where this it is impossible.

Meeting the criteria (SA1)–(SA4), (DA1) and (DA2) simultaneously might be impossible, too. Since all static criteria can be preserved by an online algorithm, the relative node stability (DA1) seems to be less important and should be relaxed in this case.

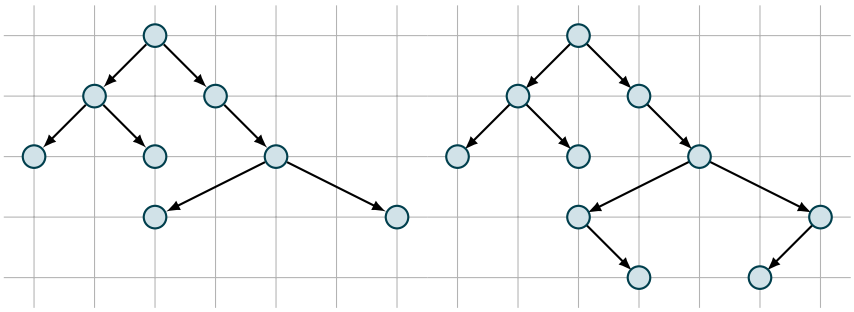


Figure 4.3: An evolving tree where the first snapshot has two isomorphic subtrees. Since one of them changes over time, the demand for relative node stability can cause the symmetry and isomorphic layout criterion for static trees to fail or results in unsatisfying wide drawings.

4.2 A Review of the Reingold-Tilford Layout Algorithm for Static Trees

An efficient and often used algorithm for static binary trees is the algorithm by Reingold and Tilford. It supports the aesthetic criteria (SA1)–(SA4). Figure 4.4 shows some drawings created with their algorithm and we can see that the drawings are quite narrow. Other approaches for unbounded or n-ary trees are often based on the same algorithmic ideas [42].

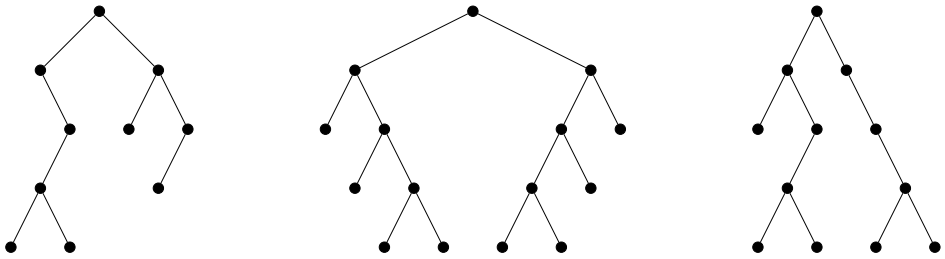


Figure 4.4: Trees drawn with the Reingold and Tilford algorithm.

Visualizing the hierarchy as demanded by (SA1) is the easiest part in the algorithm. Given a tree, we just need to determine the depth $d(v)$ of every node $v \in V$ in the tree and use it as vertical coordinate $L_y(v) = d(v)$. The more complex part is finding a satisfying horizontal layout L_x.

The initial solution is a divide-and-conquer strategy beginning at the root node. First, the layouts for the subtrees of a node are computed. Then, the neighbored subtrees are placed next to each other as close as possible. This prevents crossing edges and even supports a minimal required node distance. Lastly, the root node $v \in V$ is centered above its children $L_x(v) = \frac{1}{2}(L_x(v_\ell) + L_x(v_r))$. Figure 4.5 shows a tree before and after the placement of of its subtrees. If a node has only one child then its horizontal position gets an offset to its child's position to guarantee that the drawing respects left and right nodes (SA2).

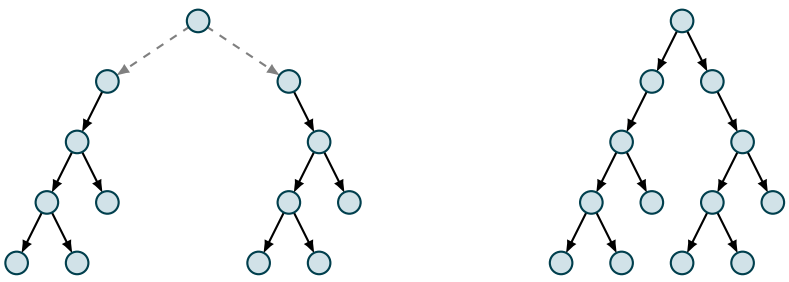


Figure 4.5: The Reingold Tilford algorithm places subtrees as close as possible to each other with respect to their borderlines.

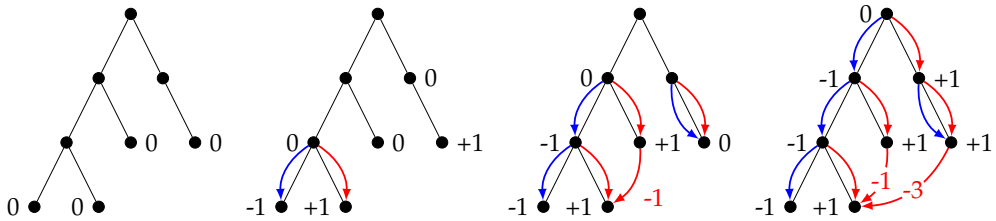


Figure 4.6: The Reingold and Tilford builds up a pointer structure and stores the shift values in a node. The absolute vertical coordinate is the sum of all shift values on the path from the root to a node.

By recursion, it follows that the leaf nodes are placed before their ancestor nodes, hence the layout apparently grows upwards. This is why isomorphic subtrees achieve the same layout. The approach can be used for trees with higher node degree too. In this case it is unimportant to have left and right nodes. Thus, for a single child w the parent v may get the same horizontal coordinate $L_x(v) = L_x(w)$.

Reingold and Tilford's algorithm has linear runtime. Note that this is not implied by the description above. The divide-and-conquer strategy leads to a tree traversal and for each node the required distance between subtrees has to be computed and all subtree nodes are shifted. Both steps, the computation of subtree distance and the shifting of subtrees, would yield to a quadratic runtime when they are implemented straight forward.

Instead of shifting whole subtrees, the original algorithm stores a relative shift at the root nodes of related subtrees, hence all horizontal positions can be accumulated in linear time by traversing the whole tree once at the end. The minimum distance between two subtrees needs to be identified efficiently, too. Given two neighbored subtrees, only the right and left most nodes in each level are required. To access these nodes efficiently a pointer structure is used. Each node gets two separate pointers, one for the left most and one for the right most node in the next level of its subtree. Follow the outermost pointers in the left and right tree the required distance can be computed for each level. Processing a node the pointers are directed to its left and right child and if subtrees have different heights then the pointer of one node in the last level has to be updated. To be precisely, for the pointers relative shifts need to be stored too. For more details see the original explanation in [34]. Figure 4.6 shows how such pointer structures might build up a whole tree.

Subsuming, the whole tree is traversed twice. First, to compute the depth, the relative shifts, and the required distances by building up and using a special pointer structure. Second, to accumulate the real horizontal coordinates. Although the computation of the minimum distance of two subtrees does not require constant time, Reingold and Tilford could showed that the given acceleration is enough to achieve a linear runtime for the whole tree. It could be shown that also the extension for non-binary trees is implementable in linear runtime [8, 42].

4.3 The Algorithm for Evolving Trees where Supergraphs are Trees

While online approaches for drawing evolving trees, like those of Moen [30] or Cohen et al. [11], produce acceptable layouts by preserving static criteria, it is desirable to profit from the setting in an offline problem where all snapshots are known. Figure 4.7 shows that even the insertion of a single node can result in multiple violations of the new criterion for relative node stability (DA1).

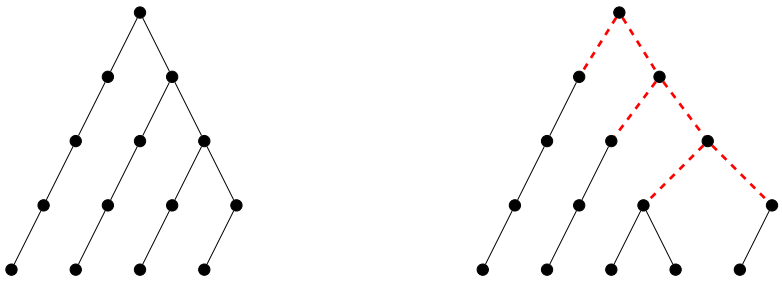


Figure 4.7: A worst case scenario where an online approach violates the relative node stability for all (dashed) edges connected to the path between the root and a new node.

In the following we design a new algorithm that takes the whole evolution of a tree into account. It tries to respect the desired aesthetic criteria (SA1)–(SA3), (DA1), and (DA2) as a weaker replacement for (SA4). In addition, it will be independent of a specified set of update operations.

We assume to have the strong precondition, that each node in an evolving tree must not change its parent node or its child position. The *child position* describes if a node is a left or right node in a binary tree or a k-th one in an n-ary tree. In such a tree only insertions or deletions of leaf nodes are allowed and it follows that the supergraph is a tree or forest.

For evolving trees with that precondition, the new Algorithm 1 produces an evolving graph layout. It uses a similar divide-and-conquer strategy as Reingold and Tilford's algorithm This is possible since the supergraph is a tree. Instead of placing single subtrees close together, for a given node, the required subtree distances in all snapshots are used to compute one common distance. Then, all subtrees at the same child position are shifted to the same position simultaneously. Nodes and subtrees in this case are like three-dimensional objects that are moved together as Figure 4.8 shows.

The new idea in the algorithm is the synchronization of the subtree distances over time. Assume we have a binary tree. For a given a node v, the algorithm computes the required distances $\delta_i^1(v)$ of its subtrees in each snapshot T_i which contains v, and uses the maximum distance as global distance $\delta^1(v)$. This prevents crossings because the distances greater than or equal to the necessary distance in each snapshot to achieve that goal. Additionally, $\delta^1(v)$ is tight since there is at least one snapshot in which this

Algorithm 1 A Reingold-Tilford inspired Layout Algorithm for Evolving Trees

procedure SIMPLEEVOLVINGTREELAYOUT($\mathcal{T} = (T_1, \ldots, T_n)$)
  Compute the supergraph $\widehat{\mathcal{T}} = (\widehat{V}, \widehat{E})$ of $\mathcal{T}$
  Compute the level $d(v)$ of each node $v \in V$ in the supergraph (tree)
  Assign the y-coordinate of each node $L_{i,y}(v) \leftarrow d(v)$ for all i where $v \in V_i$.
  Get all root nodes $R \leftarrow \bigcup_i \{v \mid v \in V_i \text{ is a root in } T_i\}$
  for all $v \in R$ **do**
    SIMPLESUBTREELAYOUT(v)
  end for
  for all $T_i \in \mathcal{T}$ **do**
    Compute the horizontal positions $L_{i,x}(v)$ by accumulation as in [34].
  end for
  return $\mathcal{L}$
end procedure
procedure SIMPLESUBTREELAYOUT(v)
  Set the initial vertical shift $s_i(v) \leftarrow 0$ for all i where $v \in V_i$.
  Let m be the number of subtrees or child positions of v.
  Compute the layouts of all subtrees in every snapshot:
  for $p := 1$ to m **do**
    Let $C_p(v)$ be all children which are at the p-th position of v:
    for all $c \in C_p(v)$ **do**
      SIMPLESUBTREELAYOUT(c)
    end for
  end for
  for $k := 2$ to m **do**
    for all $i \in \{i \mid v \in V_i\}$ **do**
      Compute the min. dist. $\delta_i^k(v)$ of the $(k-1)$-th and k-th subtree of v in T_i
    end for
    $\delta^k(v) \leftarrow \max_i\{\delta_i^k(v), \Delta_{\min}\}$
  end for
  $\delta(v) \leftarrow \sum_{j=1}^{m-1} \delta^j(v)$
  $x \leftarrow -\frac{1}{2}\delta(v)$
  for $k := 1$ to m **do**
    Shift the k-th subtree of v with x in all T_i where $v \in V_i$.
    Update the pointer structures for each T_i where $v \in V_i$ [34].
    $x \leftarrow x + \delta^k(v)$
  end for
end procedure

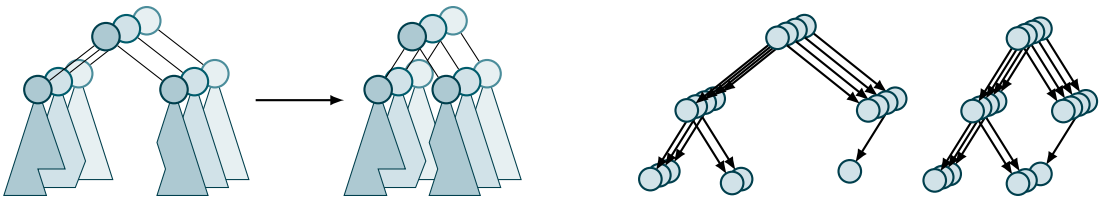


Figure 4.8: Subtrees at the same child position (e. g. left, right) are aligned and Neighbored groups of subtrees are placed as close as possible next to each other.

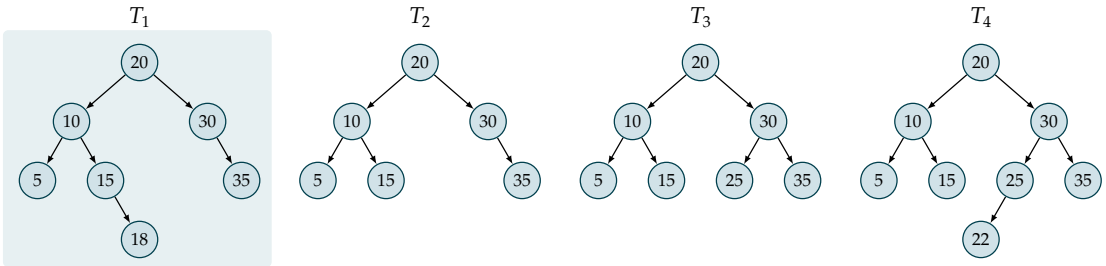


Figure 4.9: A search tree where nodes are deleted and inserted.

distance is needed. This means that the algorithm also saves space respectively width. In non-binary trees we have more than one distance of subtrees, hence for each pair of neighbored child positions $(1,2),(2,3),\ldots$, the related distances $\delta^1(v),\delta^2(v),\ldots$ are synchronized separately.

To be more efficient, the same improvements as in Reingold and Tilford's algorithm are used, too. The necessary data structures, including the pointer structures and the relative shifts have to be maintained for each snapshot independently. Note that the pointer structures must be updated after the common relative shift was computed, hence the relative shifts are applied correctly to the pointer structure.

The algorithm satisfies all desired criteria. In a binary tree the common distance δ^1 must be at least a minimum distance $\Delta_{\min} > 0$. Since the relative shifts of left most and right most children v_ℓ, v_r are $0 - \frac{1}{2}\delta(v)$ and $0 + \frac{1}{2}\delta(v)$ with $\delta(v) \geq \delta^k(v) \geq \Delta_{\min}$ (lines 30–36), v is centered above them (SA3) and the horizontal ordering (SA2) of nodes is preserved. A shift of a subtree that contains v does not change this fact.

It should be clear that a node gets the same position in each snapshot and no node will move (DA1). Subtrees with the same behavior over time are isomorphic in each snapshot and the supergraph and by the recursion, they get the same layout (DA2). Figure shows the layout of a search tree create with this new Algorithm 1.

Runtime Analysis

The algorithm has a runtime of $O(|\widehat{V}| \cdot |\mathcal{T}|)$. Remember, Reingold and Tilford's algorithm runs linear time and using it on each snapshot graph would require a time $O(|\widehat{V}| \cdot |\mathcal{T}|)$ such that we need to argue that the new algorithm is not slower.

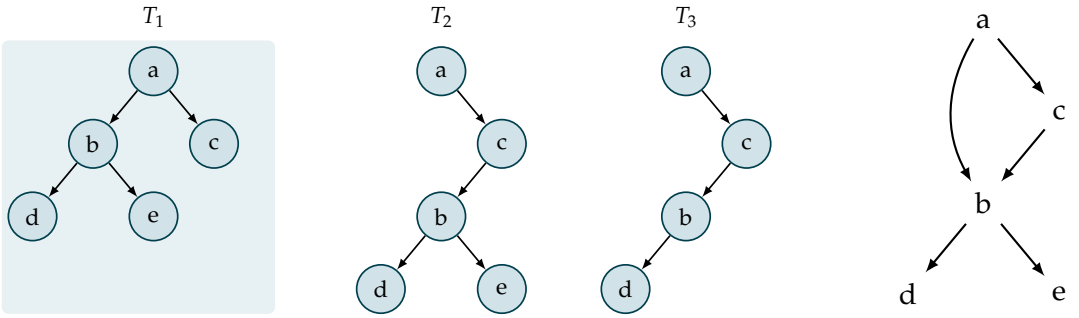


Figure 4.10: An evolving tree where nodes change their parents and its acyclic supergraph.

As in the static algorithm the relative shift distances are stored in the roots of the related subtrees and the horizontal node positions are computed in a final tree traversal in each snapshot. This requires time $O(|\widehat{V}|)$ per snapshot. The more complex part is the computation of the required distances between neighbored subtrees in the snapshots. Since the the pointer structures by Reingold and Tilford are maintained independently for each snapshot, the time to compute the distances in the snapshots and to update these pointer structure is still linear in the number of nodes for each snapshot such that the computations of all required distances and all updates have a runtime of $O(|\widehat{V}| \cdot |\mathcal{T}|)$.

The only difference to independent runs of Reingold and Tilford's algorithm is the synchronization of the required distances over all snapshots before the pointer structures are updated. This is one maximum computation for each node $v \in \widehat{V}$ and requires time $O(|\mathcal{T}|)$ per node. Thus, for the whole evolving tree we still preserve the runtime of $O(|\widehat{V}| \cdot |\mathcal{T}|)$.

4.4 The Algorithm for Evolving Trees where Supergraphs are DAGs

Obviously the new algorithm does neither require certain update operations nor analyses the transitions between two consecutive snapshots. Nevertheless, it is weaker than existent offline approaches because the precondition, that nodes must keep their child position, excludes many evolving. It is not possible to prune and regraft a subtree to another position in the tree or even to subtrees as required in balanced search trees like AVL-trees [1] or red-black trees [13].

Next, reconsider the idea of the tree traversal. We place nodes after their subtrees received a layout. For this, the horizontal layout is computed from the leaf up to the tree such that a node is reached when all subtrees have their inner layout. Algorithm 1 visits nodes post-order of a tree traversal. In fact, this is not necessary because the same result can be achieved with a different order unless the layout of all subtrees has been computed before they are adjusted and their parent node is visited. This

condition is given if nodes are visited in a topological ordering of the supergraph. Every directed acyclic graph (DAG) has such a topological order and with this insight, we can adapt the algorithm to Algorithm 2. The only difference is that this extension do not use recursion. As a result, prune and regrafting operations of subtrees as shown in Figure 4.10 are possible.

Algorithm 2 An Extended Evolving Tree Layout Algorithm

procedure EVOLVINGTREELAYOUT($\mathcal{T} = (T_1, \ldots, T_n)$)
  Compute the directed supergraph $\widehat{\mathcal{T}}$
  Check if $\widehat{\mathcal{T}}$ is acyclic, otherwise stop.
  Compute for each snapshot T_i the vertical positions $L_{i,y}$ by BFS.
  Compute a topological order over of all nodes $v \in \widehat{V}$ in the in $\widehat{\mathcal{T}}$ s. t. if $v \to w$ then v is before w.
  Reverse this order.
  for all $i := 1 \to |\widehat{V}|$ **do**
    Let $v \in V$ be the i-th last node in the reversed topological order.
    SUBTREELAYOUT(v)
  end for
end procedure
procedure SUBTREELAYOUT(v)
  for all $i := 1 \to n$ **do**
    $L_{i,x} \leftarrow 0$ if $v \in V_i$
    Let m_i be the number of children/possible children positions.
  end for
  $m \leftarrow \max_i(m_i)$ ▷ For binary trees m is either 2 or 0.
  for $j := 2 \to m$ **do**
    for $i := 1 \to n$ **do**
      Compute the minimal distance δ_i^k between the $(k-1)$-th and k-th subtree of v in T_i.
    end for
    $\delta^k(v) \leftarrow \max_{i \in \{1,\ldots,n\}}(\delta_i^k(v))$
    $\delta(v) \leftarrow \delta^k(v)$
  end for
  Adjust all subtrees with respect to the minimal distances $\delta^1(v), \delta^2(v), \ldots$.
end procedure

This modification computes a topological order of the supergraph and uses it to guarantee that a node is placed after its subtrees have received their layout. When a node is placed its position relative to is subtrees is fixed. So when the node changes its parent node or child position then the whole subtrees have to be shifted in the related snapshots. Thereby the nodes in the subtrees are not rearranged and we can guarantee that the relative node stability is still optimal because nodes only move relative to each

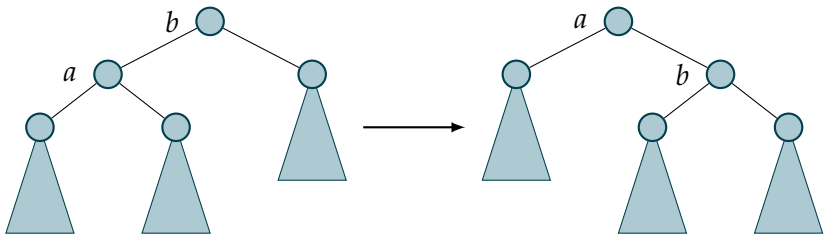


Figure 4.11: A rotation of node *a* and *b*. First *a* is an ancestor of *b* and afterwards the other way around.

other when they are not connected in one of the related snapshots.

This approach is more powerful than the first variant. We can put subtrees to different nodes in the tree unless the supergraph is a directed acyclic graphs (DAG). However, there are still those evolving trees left that have a cyclic supergraph.

4.5 The Algorithm For Evolving Trees where Supergraphs are Arbitrary

The second algorithm supports more evolving trees than the first on. Subtrees are allowed to move if they are pruned and regrafted while the relative node stability is preserved completely. There are still instances, that cannot be processed. These are all evolving trees with a cyclic supergraph.

This problem has a huge impact because graphs of many desired use-cases are still excluded. In balanced search trees like rotations as shown in Figure 4.11 are performed sometimes and these are the interesting steps where visualizations help to understand algorithms in computer science lectures. Since all rotations cause some nodes to switch their ancestor-descendant relationship, they always result in cyclic supergraphs. In addition, cyclic dependencies appear for some sequences where subtrees are pruned and regrafted, too.

Indeed, that the algorithm does not work in such cases does not imply that it is impossible to produce a layout. Nevertheless I could show that there is no other layout algorithm that always avoids relative movements between connected nodes (DA1) while it preserves the other criteria (SA1)–(SA3), and (DA2).

Theorem 4.1. *There is no layout algorithm for n-ary, binary, and unbounded evolving trees which always preserves the aesthetic criteria (SA1)–(SA3), and especially (DA1) of relative node stability and (DA2), even if only prune-and-regraft operations are allowed.*

Proof. The theorem can be proved by construction of an evolving tree as a counterexample for which it is impossible to find a layout that does not violate a criterion. The counterexample will be a binary tree, since it cover the other kinds of trees, too.

Let S be a subtree with a height of at least two. For example this could be a chain of three or more nodes. In this case the height of a tree is the number of edges on the longest path from a root to a leaf.

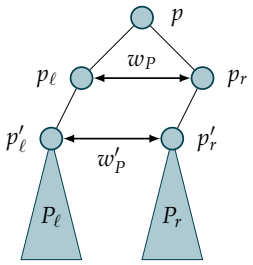


Figure 4.12: Construction of the subtree P with the isomorphic subtrees P_ℓ and P_r at the first positions of p_ℓ and p_r. w_P and w'_P denote the distances between the nodes p_ℓ and p_r, or p'_ℓ and p'_r in a layout.

Let P be a subtree with the root node p and its two children p_ℓ and p_r, which both have a first subtree in the shape of S, as shown in Figure 4.12. Let p'_ℓ and p'_r be the roots of these subtrees P_ℓ and P_r. Let Q be a second a subtree, which is isomorphic to P, and apply the naming of nodes and subtrees in Q (q_ℓ, q_r, etc.) corresponding to P (p_ℓ, p_r, etc.).

The counterexample $\mathcal{T}$ consists of three snapshots T_1, T_2, and T_3. Let T_1 be a tree that consists of P and where Q is the subtree at the second position of p_ℓ, T_2 a tree consisting of P and Q but both being subtrees of a common root, and T_3 be the similar to T_1 but in this snapshot P is in the second position of q_ℓ. Figure 4.13 shows the evolving tree $\mathcal{T}$. Since $\mathcal{T}$ contains the second snapshot T_2, the evolving tree has no rotations and can be achieved by pruning and regrafting of the subtrees P and Q.

Assume there is an a layout $\mathcal{L} = (L_1, L_2, L_3)$ for $\mathcal{T}$ preserving the desired criteria. By construction all edges in P or Q exists in each snapshot and the inner layouts of the

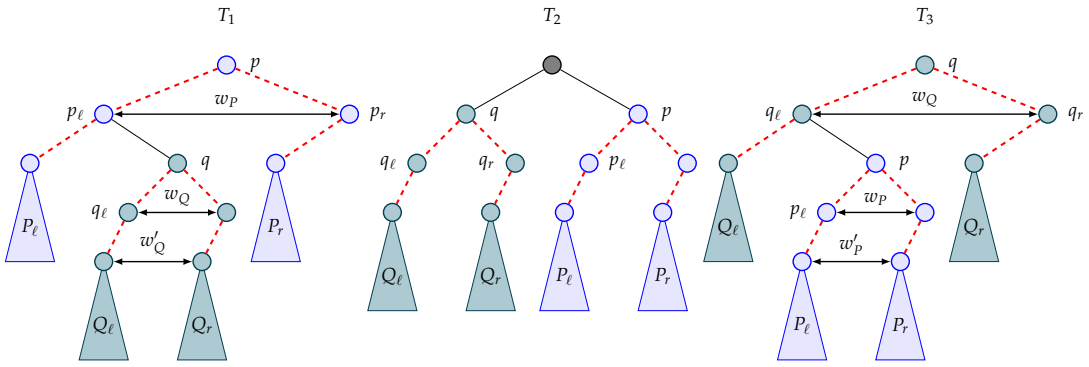


Figure 4.13: A counterexample: The dashed edges should not change over time by the relative node stability criterion (DA1).

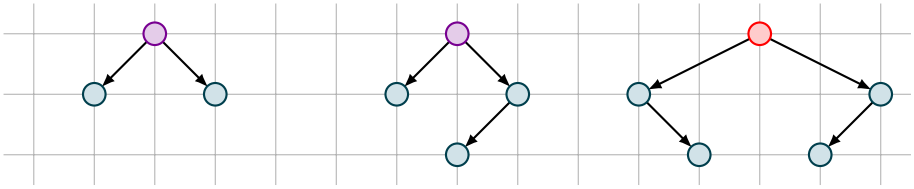


Figure 4.14: A layout created with a naive approach. The same placement can be achieved with the new layout algorithm if the root is replaced by a new node in the last snapshot.

subtrees P_ℓ, P_r, Q_ℓ, and Q_r are the same except an offset and constant over time.

The criterion (DA1) implies that the distances $w'_P = |L_{i,x}(p'_\ell) - L_{i,x}(p'_r)|$, and $w'_Q = |L_{i,x}(q'_\ell) - L_{i,x}(q'_r)|$ are constant in each snapshot T_i. Since subtrees P_ℓ, P_r are in the same shape and do not change over time, with (DA2) it follows that the distance between the right outermost nodes of P_ℓ and P_r is exactly w'_P in each level. Thus, the horizontal free width between them is at most w'_P. This implies that $w'_Q < w'_P$ because p'_ℓ and p'_r are between them in T_1. This is the same case for Q and P in T_3 such that $w'_P < w'_Q$. Finally, with $w'_P < w'_P$ we get that our assumption about $\mathcal{L}$ must be false. Hence, the contradiction proves the claim. □

The theorem implies that at least one aesthetic criterion has to be relaxed to support arbitrary evolving trees. Since the relative node stability (DA1) seems to be less important than the static criteria, we relax this criterion .

Instead of falling back into an naive or an offline approach as soon as the supergraph is cyclic, we need a strategy how we can get rid of those cycles or cyclic dependencies. Obviously, we cannot just remove edges or nodes in snapshots. Consider the evolving graph layout given in Figure 4.14. Although Algorithm 2 does not produce such a layout we can get the same result if we modify the evolving graph and replace the root node in the last snapshot by a new one. This was a transformation of a given evolving tree $\mathcal{T}$ into a new one $\mathcal{T}'$ such that the snapshots T_i and T'_i are pairwise isomorphic. If we transform an evolving tree into another evolving tree which snapshots are isomorphic with the original ones, then the layout of the modified evolving tree can be adapted for the original evolving graph. Replacing nodes by new ones can cause violations of the relative node stability but it also changes the supergraph such that we can use such transformation to get a similar evolving tree with an acyclic supergraph.

The simplest modification is to replace all nodes in every snapshot with new ones. Then, the supergraph is definitely acyclic, so the removal of cycles is always possible. Finally, we need an algorithm that converts an evolving tree into another tree and we can use the algorithm 3 as skeleton for an algorithm that supports every evolving tree. Since the relative node stability should be fulfilled as well as possible, we try to reduce the replacements of nodes to preserve the relative node stability as much as possible. A possible method is the heuristic given by Algorithm 4. It creates a new evolving

Algorithm 3 General Evolving Tree Layout Algorithm

```
1: procedure GeneralEvolvingTreeLayout(𝒯)
2:     𝒯′ ← TemporalCycleRemoval(𝒯)
3:     ℒ ← EvolvingTreeLayout(𝒯′)
4:     return ℒ
5: end procedure
```

graph and inserts all edges ordered by their snapshots successively into it until we have a *critical edge* that would create a cycle in the supergraph. Then, one node of such critical edge will be separated by replacing it with a new one in the current and all following snapshots and the heuristic continues.

The given heuristic seperates the head nodes w of critical edges (v, w). The heuristic can be changed to use the tail node or both nodes. These different *separation strategies* may result in different numbers of separated nodes. Figure 4.15 and Figure 4.16 show comparisons of evolving tree layouts generated with different separation strategies. In Figure 4.16, it seems that a minimal number of separated nodes not necessarily produce best layout. Note that the separation of a node not necessarily imply a violations of criterions.

Although it was shown with Theorem 4.1 that we cannot always avoid violations of aesthetic criteria there are also some evolving trees with cyclic supergraphs that have a layout preserving all criteria. This is also the case for the given red-black tree in Figure 4.21. It is uncertain if such layouts can be found efficiently using heuristic approaches.

Runtime Analysis

The expected runtime of the given greedy algorithm is worse than the runtime of the layout algorithm. For all edges in all snapshots it checks if the insertion creates a cycle in the supergraph. These are $O(|\mathcal{T}| \cdot |\widehat{V}|)$ edges because each snapshot is T_i a tree with $|E_i| = |V_i| - 1$ edges. The check for a cycle can be realized in time $O(|\widehat{E}|)$ using breadth first search. In the worst case each edge is critical and enforces the replacement of a node. In this case the node is replaced in all $O(|V_i|)$ edges of the currently regarded snapshot. So for each edge in a snapshot T_i the insertion requires time $O(|\widehat{E}| + |V_i|)$ such that the greedy cycle removal has a runtime of $O(|\mathcal{T}| \cdot |\widehat{V}| \cdot (|\widehat{E}| + |\widehat{V}|)) = O(|\mathcal{T}| \cdot |\widehat{V}| \cdot |\widehat{E}|)$.

Algorithm 4 A Heuristic for Temporal Cycle Removal

procedure GREEDYTEMPORALCYCLEREMOVAL($\mathcal{T}$)
  Compute the supergraph $\widehat{\mathcal{T}}$ of $\mathcal{T}$
  if $\widehat{\mathcal{T}}$ is acyclic **then**
    return $\mathcal{T}$
  else
    Let $\mathcal{T}'$ be a new evolving tree with $|\mathcal{T}|$ snapshots $T'_1, \ldots, T'_{|\mathcal{T}|}$.
    Let $\widehat{\mathcal{T}}' = (\widehat{V}' = \emptyset, \widehat{E}' = \emptyset)$ be an empty supergraph.
    Let $f \leftarrow \mathrm{id}$
    for $i = 1 \rightarrow |\mathcal{T}|$ **do**
      $T'_i \leftarrow (V'_i = f(V_i), E'_i = \emptyset)$
      Let $\overline{E}_i \leftarrow \emptyset$ be the set of inserted edges.
      for all $(v, w) \in E_i$ **do**
        if $(f(v), f(w))$ creates a cycle in $(V'_i, E'_i \cup \overline{E}_i)$ **then**
          Let $x = f(w)$, create a new node x^*, and set $f(w) := x^*$
          Replace in all edges of $\overline{E}_i$ the node x with x^*.
          Replace x with x^* in V'_i.
        else
          Insert $(f(v), f(w))$ into $\overline{E}_i$.
        end if
      end for
      $E'_i \leftarrow E'_i \cup \overline{E}_i$.
    end for
    return $\mathcal{T}'$
  end if
end procedure

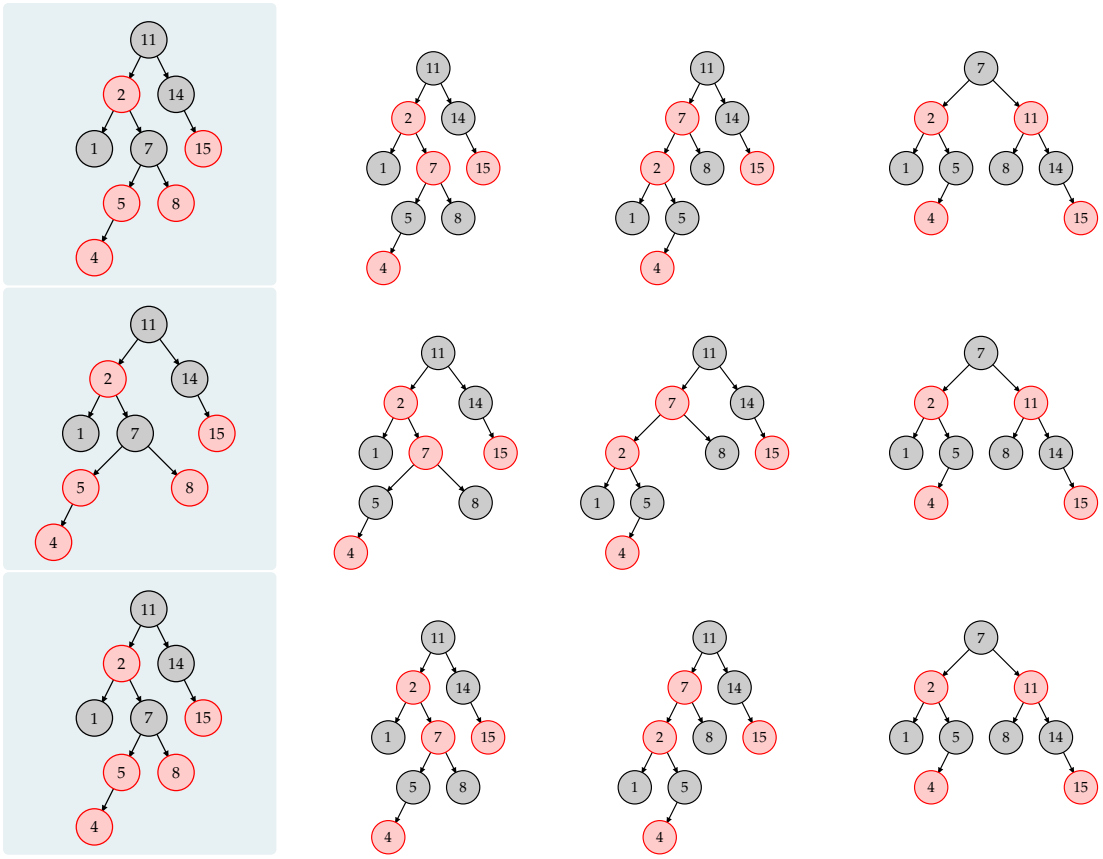


Figure 4.15: A comparison of different seperation strategies for the red black tree example. For a critical, edge the head (first and second sequence) or the tail (first and third sequence) can be separated.

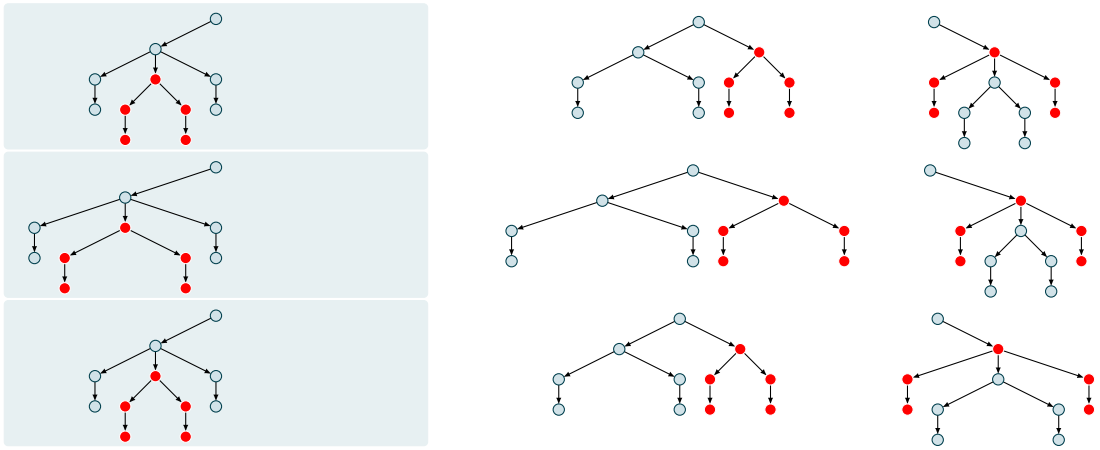


Figure 4.16: Comparing the three split strategeis for the greedy algorithm. Each configuration results in a different layout and the optimal split set induces the widest layout.

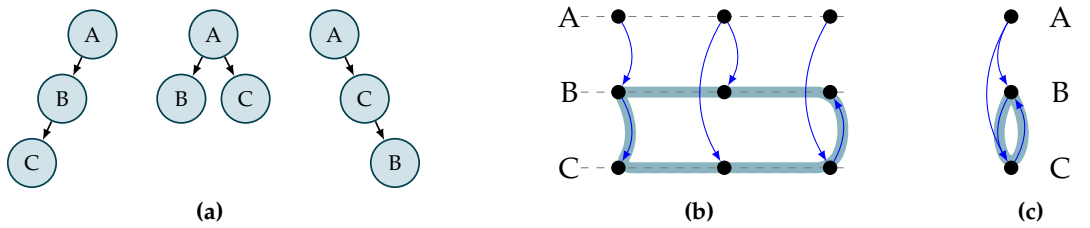


Figure 4.17: An evolving tree (a) with a cyclic dependency and its time-line representation (b) with the supergraph of the tree (c). The cyclic dependency is highlighted.

4.6 The Complexity of Temporal Cycle Removal

Since the replacement of all nodes in each snapshot by new nodes results in the same layouts we would achieve by using a naive algorithm, it should be clear that the relative node stability gets worse the more nodes are replaced. The given algorithm was only a heuristic but there might be an algorithm which reduces the modifications of an original evolving tree to a minimum. In this section we investigate the complexity of the cycle removal problem in the supergraph and we will see that this problem is NP-complete, hence the use of a heuristic is acceptable.

For better analysis and a closer look on cyclic dependencies we use a time-line representation as two dimensional visualization of an evolving tree [2, 24]. All snapshots are drawn separately in columns and nodes representing the same node are connected by a line. As the child positions do not influence whether the supergraph is cyclic not they are not visualized. Figure 4.17 shows an example of a cyclic evolving tree with its time-line representation.

This representation is useful to analyze when which nodes are involved in a cyclic dependency. It can be seen as an own static graph itself with mixed edges. If there is a cycle in the supergraph then, there is also a cycle in this time-line graph. For such a cycle an undirected edge may be used in both directions, but only one time.

Replacing a node in a certain snapshot and in all following snapshots means for the time-line representation that an undirected edge between two snapshots will be removed as in the new node is a different one and corresponds to a new time-line. Formally, we can describe a transformation of an evolving tree into a similar evolving tree with a set of nodes and snapshots where nodes are replaced.

Definition 4.2 (Temporal Separation)**.** *Let $\mathcal{G}$ be an evolving graph. A* temporal separation *is a set of index-vertex pairs $S \subseteq V \times (\{1, \dots, |\mathcal{G}| - 1\})$.*

A temporal separation induces an isomorphic evolving tree as we suggested before. We denote the induced evolving graph with $\mathcal{G}[S] = (G_1[S], G_1[S], \dots)$. Each element $(v, i) \in S$ implies that the node v is replaced in all snapshots starting from G_i in $\mathcal{G}[S]$. Figure 4.18 shows how a given temporal separation induces a new evolving

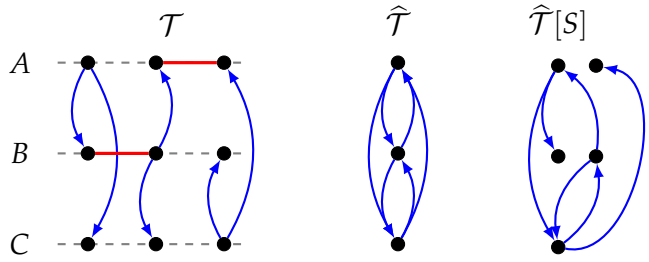


Figure 4.18: The timeline representation of an evolving tree $\mathcal{T}$, its supergraph $\widehat{\mathcal{T}}$, a separation $S = \{(A,2),(B,1)\}$, and the supergraph $\widehat{\mathcal{T}}[S]$ of the resulting isomorphic tree $\mathcal{T}[S]$. In the induced evolving tree $\mathcal{T}$ the nodes after the red lines are replaced by new nodes.

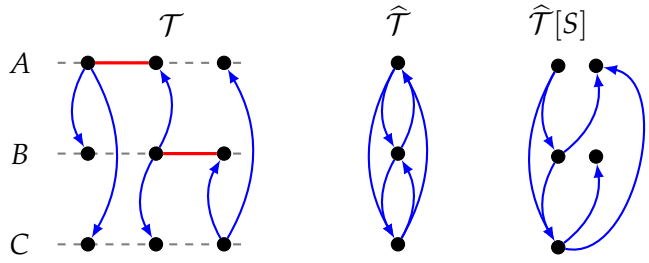


Figure 4.19: The time-line representation of an evolving tree $\mathcal{T}$, its snapshotgraph $\widehat{\mathcal{T}}$, a $S = \{(A,1),(B,2)\}$ of $\mathcal{T}$ (red lines), and the resulting acyclic supergraph $\widehat{\mathcal{T}}[S]$.

tree. In this example we can see the supergraph changes but is still cyclic. Since we want to enforce an acyclic supergraph we are interested in special separation.

Definition 4.3 (Temporal Cycle Decomposition)**.** *Given an evolving tree $\mathcal{T}$, a* temporal cycle decomposition for $\mathcal{T}$ *is a temporal separation S for which the induced supergraph $\widehat{\mathcal{T}}[S]$ is acyclic.*

Figure 4.19 shows the same evolving graph but with a different temporal separation which is a temporal cycle decomposition. To improve the coherence or stability in a layout for $\mathcal{T}$ a temporal cycle decomposition S should be as small as possible, which is an optimization problem. Unfortunately this is a hard problem as I could show that the related decision problem is NP-complete.

Definition 4.4. *The problem* TEMPORALCYCLEDECOMPOSABLE (TCD) *contains all tuples $(\mathcal{T},k)$ of evolving trees $\mathcal{T}$ if there is a temporal cycle decomposition S for $\mathcal{T}$ of size $|S| \leq k$.*

Theorem 4.5 (NP-completeness of TCD)**.** *The decision problem* TEMPORALCYCLEDECOMPOSABLE *is* NP*-complete.*

Proof. Since the test for a cyclic graph is in polynomial time and the maximum size of a temporal separation is at most $|\mathcal{T}| \cdot |\widehat{V}|$, the problem must be in NP. To show hardness

we reduce the NP-complete problem VERTEXCOVER to the new problem TCD. The problem VERTEXCOVER decides for an undirected graph G and a given number k if it is possible to find a set of k nodes in G such that each edge in G is incident to at least one node of this set. As reduction function we describe a transformation of instances for the VERTEXCOVER problem into our problem:

Let $(G = (V, E), k)$ be an input for the VERTEXCOVER problem with a graph G and a number k. We need to generate an instance for TCD and to be precisely an evolving tree $\mathcal{T}$ and a number k'. For each $v \in V$ let $v_{\text{in}}, v_{\text{out}}$ be two new nodes and $\widehat{V} = \{v_{\text{in}}, v_{\text{out}} \mid v \in V\}$ be the set of nodes in the evolving tree. For each node $v_i \in \{v_1, \dots, v_n\} = V$, let $T_i = (V_i = \widehat{V}, E_i)$ be a snapshot where $E_i = \{(v_{i,\text{out}}, w_{\text{in}}) \mid \{v_i, w\} \in E\}$ contains a directed outgoing edges from $v_{i,\text{out}}$ to w_{in} for each node w that is connected with v_i in G. And let $T_{|V|+1} = (\widehat{V}, E_{|V|+1})$ be a last snapshot whit an edge $(v_{\text{in}}, v_{\text{out}}) \in E_{n+1}$ for each node $v \in V$. By construction each snapshot is a tree or forest.

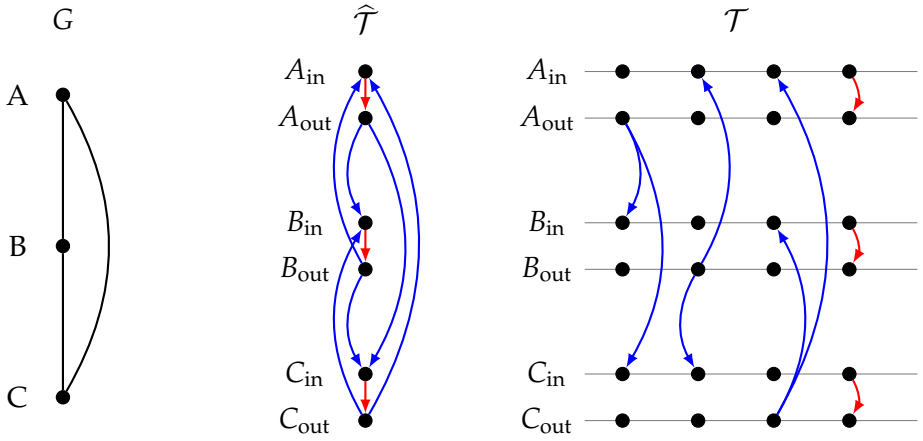


Figure 4.20: The reduction of VERTEXCOVER on TEMPORALCYCLEDECOMPOSABLE. A given graph G gets transformed into the evolving tree $\mathcal{T}$. Each node v maps to the nodes v_{in} and v_{out} in the evolving tree and an edge $(v_{\text{in}}, v_{\text{out}})$ (red). Each undirected edge $\{v, w\}$ in the origin graph G corresponds the two edges $(v_{\text{out}}, w_{\text{in}})$ and $(w_{\text{out}}, v_{\text{in}})$ (blue)

Finally, let f with $f(G, k) := (\mathcal{T} = (T_1, \dots, T_n, T_{n+1}), k)$ be the reduction function. The described transformation can be done in polynomial time. In Figure 4.20 we can see an example for this reduction and the related supergraph $\widehat{\mathcal{T}} = (\widehat{V}, \widehat{E})$. Note that for each edge in G we achieve exactly one atomic cycle in the supergraph and all cycles in the supergraph are alternating paths of in-out (red) and out-in (blue) edges. Indeed, this function does not create an evolving forest instead of a tree but this can be fixed by insertion of one new common root node that does not affect the cycles in the supergraph.

Now, we have to prove the correctness of this reduction. For that reason we show that $(G, k) \in$ VERTEXCOVER if and only if $(\mathcal{T}, k) = f(G, k) \in$ TCD. Let (G, k) be an

instance of VERTEXCOVER then there is a vertex cover S of size $|S| \leq k$ for the graph $G = (V, E)$. The evolving tree $\mathcal{T}$ of $f(G, k) = (\mathcal{T}, k)$ has $|V| + 1$ snapshots. We define the temporal separation $R = \{(v_{\text{in}}, n) \mid v \in S\}$. Obviously it holds that $|R| = |S| \leq k$. It follows that R is a temporal cycle decomposition for $\mathcal{T}$. By construction no in-node v_{in} has incoming edges in G_{n+1} and all out-nodes have exactly one in-node. A given vertex v_{in} can be divided by R into at most two nodes v_{in} and v'_{in}. This implies that if a node v is in the vertex cover S then in $\mathcal{T}[R]$ neither v_{in} nor the node v_{out} can be on a cycle. If there is a cycle left then there is an alternating path in the supergraph with at least two in-out-edges. Those correspond to connected nodes x, y in G. As the in-out-edges are on that path, neither (x, n) nor (y, n) is in the separation R. Hence, x, y are both not in the vertex cover S. Since x and y are connected S cannot be a valid vertex cover and it follows that $\widehat{G}[R]$ is cycle free.

Now let $f(G, k) = (\mathcal{T}, k) \in$ TCD. This means we have a separation R of size k such that $\widehat{\mathcal{T}}[R]$ is cycle free. We can replace all elements (v_{in}, j) and (v_{out}, j) in R by (v_{in}, n) and receive another separation R' by that with $|R'| \leq |R|$. R' also decomposes all cycles in $\mathcal{T}$ because all cycles contain an in-out edge of the last snapshot. Let $S = \{v \in V_G \mid (v_{\text{in}}, n-1) \in R'\}$. Then S is a vertex cover in G. If there is an edge $\{u, v\}$ in G not covered by S then (v_{in}, n) and (u_{in}, n) are not in R'. This would imply the contradiction that R' is a complete separation for $\mathcal{T}$ because cycle $u_{\text{in}}, u_{\text{out}}, v_{\text{in}}, v_{\text{out}}, u_{\text{in}}$ would still be in the supergraph. Hence, follows the claim. □

The reduction function in the proof produces an evolving graph with many snapshots. In practice, for example in a lecture, we can rather expect more nodes instead of having a long sequence of trees. If the hardness only depends on the number of snapshots it might be a feasible problem in practice. However I could show that the problem remains hard even when the number of snapshots is limited.

Theorem 4.6. *The bounded problem* TEMPORALCYCLEDECOMPOSABLE *is* NP*-complete, even if the number of snapshots is bounded by* $|\mathcal{T}| \leq r$ *with* $r \geq 4$.

Proof. To prove this stronger statement we will use a similar reduction as for the unbounded case. The only reason why we need multiple snapshots is that each snapshot graph must be a rooted tree. Therefore it is not allowed to have a node with more than one incoming edge in the same snapshot. So we chose very freely n different snapshots to accomplish this condition without regard to the required size. It should be clear that if we allow directed acyclic graphs instead of trees exactly two snapshots are enough for the proof because on can be used for the out-in-edges and the other one for the in-out-edges.

It is proven that even for cubic graphs with a node degree of three VERTEXCOVER is NP-complete. We use the same reduction as before but show that it is always possible to reduce the number of required snapshots to four such that all out-in edges are in T_1, T_2, or T_3 and the others are all in T_4.

If we miss the last snapshot in the evolving graph $\mathcal{T}$ by the original reduction $f(G, k)$ then the supergraph is acyclic because it only contains the out-in edges. This

means edges can be exchanged between the first $n = |V|$ snapshot without loosing this condition. Since every node $v \in V$ of an undirected cubic graph $G = (V, E)$ has degree of at most three, the number of incoming edges of the node v_{in} in the supergraph $\widehat{f(G,k)}$ is at most three, too. Thus, it is possible to redistribute all incoming edges in $T_1, \ldots, T_n$ of a node v_{in} into the three snaphots T'_1, T'_2, T'_3 and let $T'_4 = T_{n+1}$. Then the supergraph of $\mathcal{T}' = (T'_1, \ldots, T'_4)$ is the same as for $\mathcal{T}$ and because every snapshot is acyclic and no node with an ingoing degree of more than one exists, $\mathcal{T}'$ is an evolving tree, too. The correctness of the reduction follows by the same arguments as of the previous proof. □

Note that the supergraph we achieve in both proofs is the same. It is also the same directed acyclic graph we would achieve by the VERTEXCOVER reduction on the FEEDBACKARCSET problem by Karp [26]. This problem asks for a directed graph if there is a set of k edges or arcs such that the graph without these edges is acyclic. Although there are exact FPT algorithms for the FEEDBACKARCSET problem they cannot be used because it is not allowed to remove edges in the evolving tree.

Remember that the seperation of nodes nodes is the same as removing undirected edges in the time-line representation. With this perspective one could reduce the TCDproblem to the weighted FEEDBACKARCSET problem with mixed edges. The removal of directed edges could be permitted by setting different weight for directed and undirected edges. In [5] Bonsma et al presented an FPT algorithm for the FEEDBACKVERTEXSET in mixed graphs and they provided a reduction for the weighted FEEDBACKARCSET is proposed by the authors.

The new algorithm and its heuristic presented have been implemented and were used for several examples. Since the heuristic approach produced acceptable layouts and the fact that a minimal temporal cycle resolution must not necessarily produce optimal layouts, there might be other possibilities where the algorithm could be improved. Maybe some avoidable violations that occur could be reduced for a given temporal separation. Figure 4.21 shows a layout for the previously used red-black tree example where no violation of the aesthetic criteria occurs, although it has a cyclic supergraph.

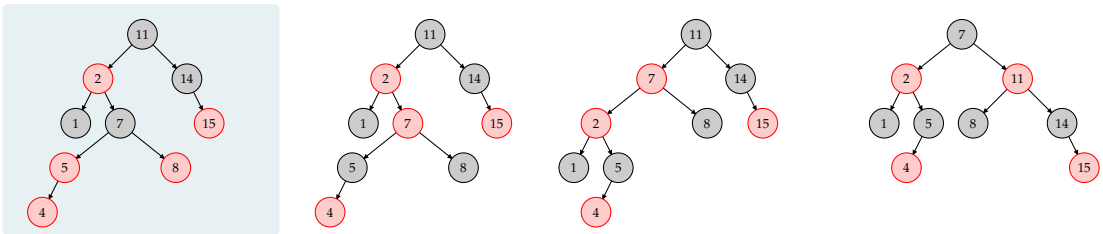


Figure 4.21: An animation of a red-black tree without a violation of the aesthetic criteria.

5 Implementation: Drawing Evolving Graphs in Ti*k*Z

TEX is a typesetting system developed by Donald E. Knuth and well known for its typesetting of mathematical formulas. It is often used by researchers, especially mathematicians, physicists, and computer scientists. For creating documents, TEX provides a powerful macro-based markup language and TEX transforms the document markup into a desired format like Postscript or PDF.

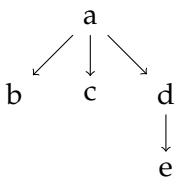
There are several packages written for TEX by an active community providing collections of macros for special purposes. One of those packages is Ti*k*Z. It was developed by Till Tantau and allows the creation of vector graphics [41]. Creating vector graphics with Ti*k*Z, or TEX in general, can be an elaborating work: When we want to draw a graph, we have to define the whole layout and all node positions ourselves. However, Ti*k*Z contains a graph drawing engine which provides a simplified graph syntax and allows to select a layout algorithm. The graph drawing engine and its layout algorithms are completely written in the programming language Lua, and require the use of a specific TEX variant, called LuaTEX, to compile the TEX documents.

In this chapter we review the graph drawing engine of Ti*k*Z. I introduce how evolving graphs can be defined in the graph syntax and how the algorithms presented in this thesis can be used. An overview of the structure and concepts of the prototype implementation is given, and we conclude with a minimal case example for the creation of an animated scalable vector graphic from a whole TEX document.

5.1 A Review of Static Graph Drawing in Ti*k*Z

Ti*k*Z allows the creation of graphs with the `\graph` macro. This macro expects one key specifying which algorithm is used for the layout and allows the description of a graph in a short syntax as you can see in Figure 5.1. The example shows a tree and its Ti*k*Z code.

In the graph syntax nodes are defined by unique names. An edge between two nodes v_1, v_2 can be defined by writing v_1 `->` v_2 for a directed, or v_1 `--` v_2 for an undirected edge. Curly brackets in the code may be used when a single node should be connected with multiple nodes and they simplify the definition of trees. For binary



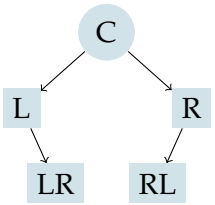

```
1 \tikz\graph[tree layout] {
2   a -> {b, c, d -> e}
3 };
```

Figure 5.1: Creating a simple drawing of a tree in Ti*k*Z.

trees, the children can be separated by a comma and if there are nodes with only one child we leave an empty space for the missing node as seen in Figure 5.2 in line 5 and 6. For users who are familiar with Ti*k*Z it should be mentioned that styling nodes and edges is possible in the graph syntax. For more details see the official manual [40].

The implemented layout algorithms are accessible with different keys. The keys `tree layout` and `binary tree layout` in the examples select a Reingold–Tilford based algorithm for non-binary and binary trees. There are also keys for layered (`layered layout`) and force-directed algorithms (`spring layout`). When a graph is defined in Ti*k*Z the graph drawing engine is started and selects the layout algorithm by looking up the key. Data structures to represent the graph or the directed graph are created and passed to the layout algorithm. The whole layout process consists of several stages and possibly some pre- or post-processing steps like the computation of a spanning tree or an edge routing algorithm are executed, too.

Nodes as Ti*k*Z objects are created before a layout algorithm starts, because their properties like the size are required for the layout algorithm, but edges are created afterwards since they depend on the node positions. A special feature in the drawing of edges in Ti*k*Z is the computation of the tail and head positions. The drawing of an edge is complex as a node is no infinitesimal small dot but at least a circle or rectangular text field. An edge should be drawn between the shapes of nodes rather than between their center positions. Ti*k*Z views an edge that connects the center positions and finally computes where the edge intersects with the node's shape as visualized in Figure 5.3.




```
1 \tikz\graph[
2   binary tree layout,
3   nodes={rectangle,...}] {
4   C[circle] -> {
5     L -> {   ,LR},
6     R -> {RL,   }
7   }
8 };
```

Figure 5.2: A binary tree with some style options in Ti*k*Z.

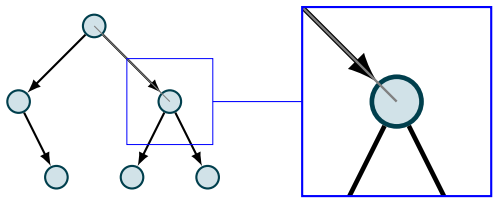
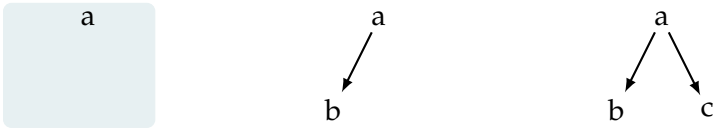


Figure 5.3: A straight line as edge connecting two nodes. The end of the edge are the intersection points of the straight line and the objects representing the nodes.

5.2 Drawing Evolving Graphs in Ti*k*Z

For evolving graphs, graph drawing is more complex because originally neither Ti*k*Z nor the graph drawing engine was designed for that purpose. This is resulting in two main problems that have been solved during the development of this thesis.

First, we need a way of specifying evolving graphs. It turns out that it is possible to describe those graphs in the current syntax with some new options. The main idea is to represent a node by different Ti*k*Z nodes, one for each snapshot, such that they can be cross-fade during an animation. This has the advantage that color, shape, or other properties can be defined as usual for every snapshot independently. Since an algorithm needs to identify which Ti*k*Z nodes correspond to the same graph node, called *supernode*, or the same snapshot, the user has to declare this for each Ti*k*Z node by using the new options `supernode` and `snapshot`. With the right styles and settings in Ti*k*Z these can be declared indirectly with another option (`when`) such that an evolving graph can be defined quite short as shown in Figure 5.4. The option passed to the or `when` or the `snapshot` key is a time in seconds that specifies the time mapping for an animation.



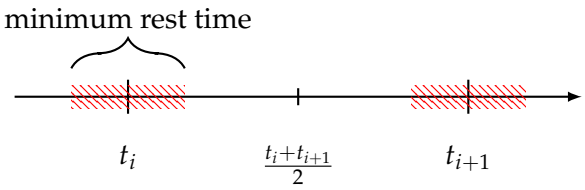

```
\tikz\graph[animated binary tree layout,
            auto supernode] {
  {[when=1] a},
  {[when=2] a->{b,}},
  {[when=3] a->{b,c}},
};
```

Figure 5.4: An example of an evolving tree written down in the graph syntax. The `when` key at the beginning of a scope assigns all Ti*k*Znodes in the same scope to the same snapshot at the given time.

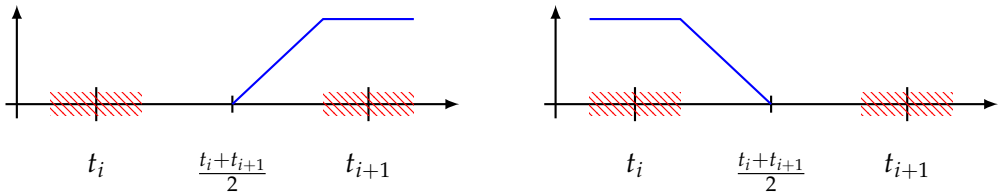
As second problem algorithms have to produce animations. In a current version Ti*k*Z supports the creation of animations for the graphic format SVG. The specification of SVG allows animations and with them one can manipulate certain properties of objects like the opacity or coordinates. Hence, it is possible to fade or move objects. The graph drawing engine allows layout algorithm to attach such animations to nodes and edges.

Since all nodes and edges are represented by different Ti*k*Z objects, these must be shown or hidden at the right time. Consider a time mapping τ. All Ti*k*Z nodes mapped to the i-th snapshot need to be visible at time $t_i = \tau(i)$. It needs to be controlled when they appear, disappear, or are cross-faded depending on certain conditions. If two Ti*k*Z nodes of the same graph node in two consecutive snapshots have different positions, the motion needs to be added to the animation attributes, too. All animations are interpolated linear.

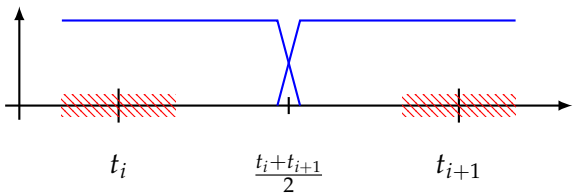
To ensure that a viewer has a chance to see and understand each single state of a graph as whole, a special break or rest time as a duration around the time $t_i = \tau(i)$ of a snapshot in which no animations occur is needed:



Appearing, or disappearing nodes increase or decrease their opacity between the mid-time $(t_i + t_{i+1})/2$ and the rest time of the snapshot in which they exist.



If a node exists in two consecutive snapshots its Ti*k*Z nodes are cross-faded by changing the opacity for a short time at the mid time between two snapshots.



Edges are faded in and out in the same way as nodes. When an appearing or disappearing edge connects two nodes that are moved during a transition it is invisible

once the nodes are in motion or get visible when their motion stopped. If an edge exist in two consecutive snapshots, then it will be visible for the whole transition even when the nodes are moved.

One unsolved problem appears when connected nodes are moved relative to each other. In these case, edges change their directions and length between two snapshots. Since consecutive Ti*k*Z edges are completely different in this case, we get rough effects and invalid drawings during the transition by cross-fading them as seen in Figure 5.5. For smoother transitions it is necessary to morph the geometric Ti*k*Z object of an edge. Except by using approximations, it is unclear how this could be realized in Ti*k*Z since the end points of the edges neither move on straight lines nor change their coordinates linearly. This is another argument for the criterion of relative node stability.

All algorithms that were presented in detail have been implemented. The foresighted layout and the improved time-dependent force-directed algorithm presented in Chapter 3 are based on Bruhns's [7] implementation of the static force-directed algorithm by Reingold and Fruchterman. They are available with the algorithm keys `foresighted layout` and `evolving force layout`. The new offline algorithm developed in Chapter 4 can be used with the key `animated binary tree layout` for binary trees and `animated tree layout` for non-binary trees.

5.3 Structure of the Prototype for Animating Graphs in Ti*k*Z

As the graph drawing engine has no special support for evolving graphs and since graph nodes are represented by multiple Ti*k*Z nodes, layout algorithms for evolving graphs have to pre-process and maintain all required temporal data structures, like the supergraph, themselves. Those algorithms also need to implement a number of standard preprocessing steps themselves that are normally taken care by the graph drawing engine, such as the computation of connected components or spanning trees. The reason is that for the graph drawing engine an evolving graph is a static graph consisting of several independent components.

The generation of the supergraph and generation of animations commands are outsourced into single components since these tasks are needed in each algorithm for evolving graphs. Figure 5.6 shows the embedding of such an algorithm in the architecture of the graph drawing engine.

The computation of the whole supergraph analyses the raw input graph and distributes the Ti*k*Z nodes by their options into single snapshot graphs and create the supergraph with new nodes. Each algorithm for evolving graphs generates the supergraph once the algorithm started in its `run()` function:

```
1 function SimpleLayout:run()
2   self.supergraph= Supergraph.generateSupergraph(self.digraph)
3   ...
4 end
```

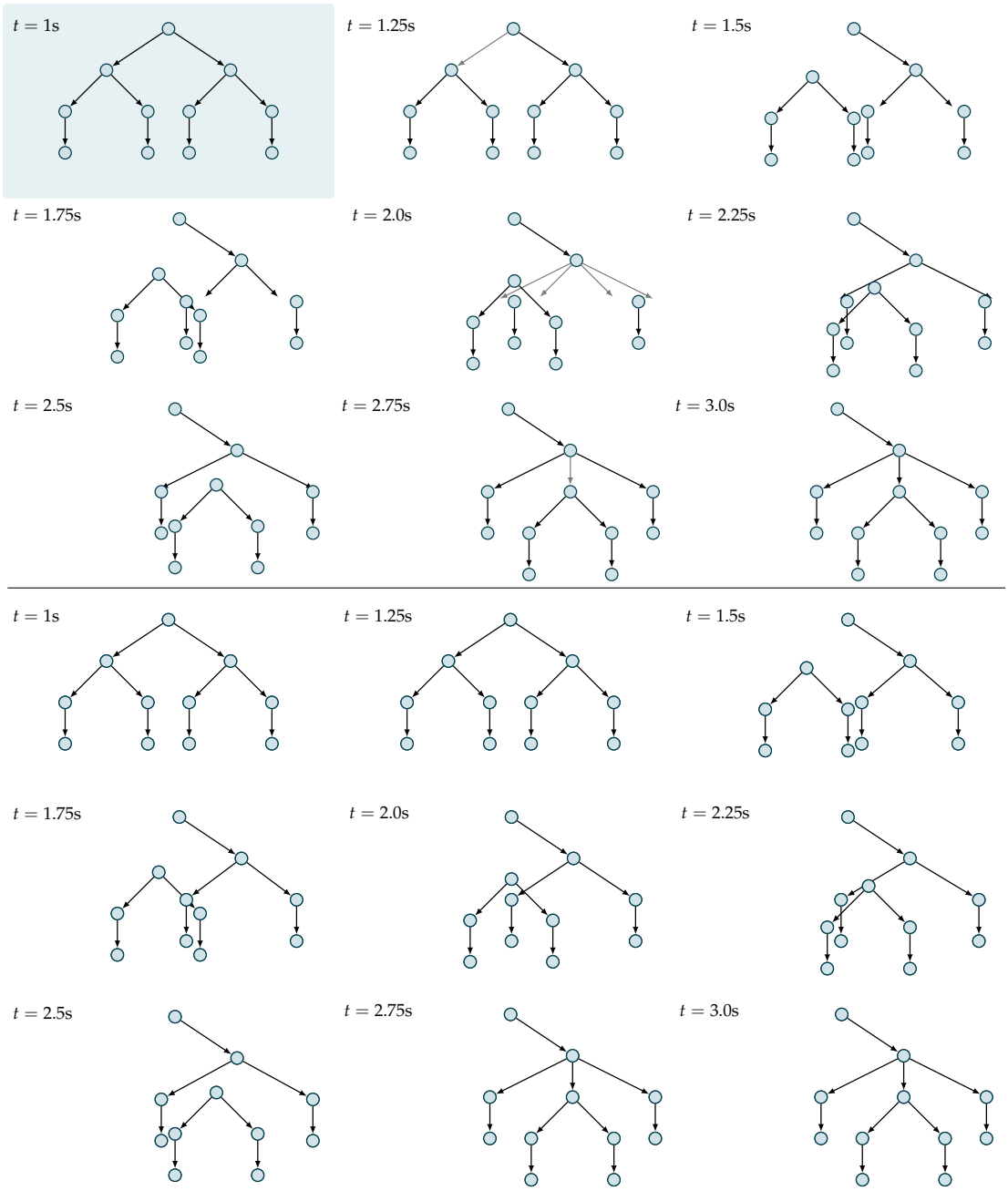


Figure 5.5: An evolving graph with a violation of the relative node stability. The upper sequence has been created by the new prototype and we can see undesirable effects and broken edges for $t \in [1.5\text{s}, 1.75\text{s}]$. The second sequence would be the preferred choice and shows how it would look like if the transformation of edges can be realized.

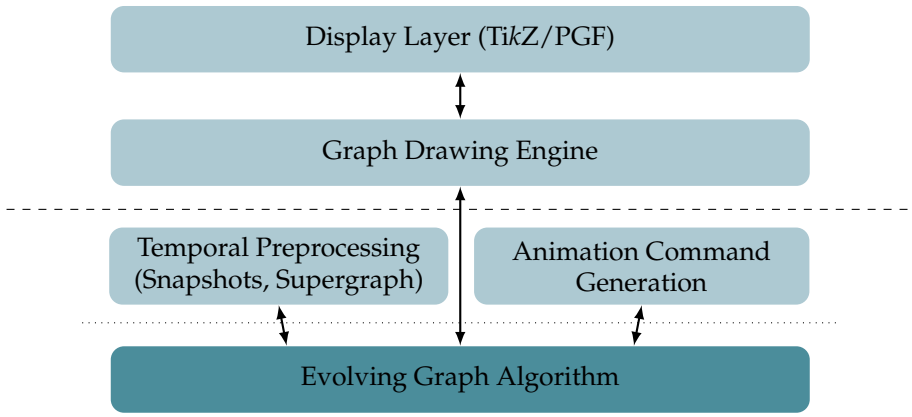


Figure 5.6: The embedding of the prototype into the existing architecture of the graph drawing engine. An algorithm for an evolving graph is called as a usual layout algorithm by the TikZ-Graph-Drawing-Engine and gets a static graph with options for each node. All temporal pre- and post-processing has to be maintained by the algorithm using these options. The processing steps could be separated into a special layer.

After its generation, the new supergraph object provides several methods for navigating through the evolving graph. Each snapshot is stored as separate graph, and given a snapshot and a graph node, the related TikZ node can be determined. The final positions and animation attributes must be assigned to the TikZ nodes, as the graph drawing engine is not aware of the whole supergraph. The following code snippet could be used in a foresighted layout algorithm that places all nodes on a circle with a given radius:

```
local supergraph = self.supergraph
local n = #self.supergraph.vertices -- no. of graph nodes
for i, supernode in ipairs(supergraph.vertices) do
  local deg = math.pi * 2 * i/n
  local x   = radius * math.cos(deg)
  local y   = radius * math.sin(deg)
  -- for all snapshots where in which the supernode exists
  for _, s in ipairs(supergraph:getSnapshots(supernode)) do
    -- apply coordinates to the node:
    local node = supergraph:getSnapshotVertex(supernode,s)
    node.pos.x = x
    node.pos.y = y
  end
end
```

When the positions of all nodes are determined, the animation entries can be appended to the TikZ nodes. They are faded in or out, and probably moved between two snapshots. All these animations are generated by a special algorithm which is

registered to a new phase `evolving graph animation`. An evolving graph algorithm can request for such an algorithm to get the default implementation or another algorithm that could be implemented.

```
function SimpleLayout:run()
  ...
  -- get a registered graph animation algorithm:
  local ga_class = self.digraph.options.algorithm_phases['
     evolving graph animation']
  -- create and run the instance of this algorithm on all
     nodes:
  ga_class.new({digraph = self.digraph, supergraph = self.
     supergraph, ... }):run()
end
```

The possibility of declaring such a phase is already provided by the graph drawing engine and allows to have exchangeable algorithms for subproblems. With this special phase it is possible that a user can choose different animation strategies if they are implemented. For example in a tree it might be a better solution that subtrees are not moved on a straight line to prevent crossing edges and maybe some acceleration or ease effects instead of constant speed might improve the quality of the animation.

5.4 A Real-life Example: From TEX Code to an Animated Scalable Vector Graphic

In this chapter, it was suggested that TikZ can now be used to easily create animations of evolving graphs. In this case we need with SVG a target format that supports animations. To demonstrate how animated graph drawing in TikZ can be used, I present all steps that are required to create a complete animated SVG of the example previously shown in Figure 5.4:

First, we have to create a new TEX document, named `MyAnimatedGraph.tex`, and write down the following TEX code.

```
\documentclass[dvisvgm]{standalone}
\usepackage{tikz}
\usetikzlibrary{graphs}
\usetikzlibrary{animations}
\usetikzlibrary{graphdrawing}
\usetikzlibrary{graphdrawing.evolving}
\usegdlibrary{evolving}
\begin{document}
\tikz\graph[animated binary tree layout,
            auto supernode] {
  {[when=1] a},
```

```
12   {[when=2] a->{b,}},
13   {[when=3] a->{b,c}},
14 };
15 \end{document}
```

The Ti*k*Z code of the evolving tree example can be found in the lines 9–14. In the preamble above (lines 1–7) all packages and Ti*k*Z-libraries that we need are included. An important difference to other TEX documents can be found in line 1. We specify that the that the `dvisvgm` driver is used since the final output format will be SVG.

As we use the graph drawing engine of Ti*k*Z, we use LuaLATEX to process our document by typing the following on the command line:

```
1 lualatex --output-format=dvi MyAnimatedGraph
```

This will create a file `MyAnimatedGraph.dvi`. Still, this is no animation, but this step is required because there is currently no TEX variant producing SVG-files directly. Now, we can use the `dvisvgm` driver to generate the animation:

```
1 dvisvgm  MyAnimatedGraph
```

This produces the final animation into the file `MyAnimatedGraph.svg`. We can view this animation by using a SVG renderer, for example a modern web web browser and we are finished. This minimal example can be used for any other evolving tree when just replace the specification.

6 Conclusion and Outlook

In computer science, we often have trees or graph structures that change over time. Animations help viewers to understand what they describe and what happens with them. We investigated the drawing of binary or n-ary evolving trees in the setting of an offline problem with the objective to develop algorithms that can be used to create useful animations in Ti*k*Z. While other approaches for evolving trees provide specific but restricted sets of update operations or allowed changes on trees, a new solution for arbitrary evolving trees has been developed.

In Chapter 2, we reviewed the terminology of evolving graphs, presented the concepts of update and state based evolving graph models, and described the problem of drawing evolving graphs. In Chapter 3, general offline approaches were reviewed and turned out to be unpractical for trees if nodes are not allowed to moved. As a result, existent online approaches which are designed for trees, like those by Moen or Cohen et al., still achieve better results than the generic offline approaches.

The essence of this thesis was the new algorithm I designed in Chapter 4 as a first offline approach for evolving trees that profits from the setting in offline problems by taking the whole evolution of a tree into consideration, thereby reducing unnecessary movements of nodes. It has the ability to produce more stable evolving layouts than online algorithms. In addition, the algorithm runs on arbitrary evolving trees such that multiple changes at the same time are possible in contrast to other algorithms that expect specific update operations. For the algorithm we identified aesthetic criteria that seem to be reasonable for animations, and I was able to show that preserving them simultaneously is impossible sometimes. The minimization of violations of the criterion for relative node stability leaded to a new NP-complete problem so that a heuristic approach was used in the algorithm. Since the replacement or separation of a node over time does not necessarily imply relative movements of nodes, it remains an open question how well the heuristic approach suppress motions and which influence different separation strategies have to the quality of an animation. If there are only a few snapshots for a short time that cause wider subtree distances in most snapshots, it is possible that in this case the relative node stability is self-defeating and the effect confuses a viewer. In such cases it might be more reasonable to enforce or allow some violations. This might be investigated in an experimental user study.

All presented algorithms, the new one and the other offline approaches, have been tested and are implemented in a new prototype as a proof of concept for animated

graph drawing in Ti*k*Z. This prototype, presented in Chapter 5, provides new possibilities in the visualization of trees and might help to improve lectures in the future and was used to create all examples in this thesis. Its architecture allows other people to implement new algorithms for evolving graphs. This prototype has to be integrated in the Ti*k*Z in the future and there are some open tasks left that were not part of this thesis. This includes the motion planning or improvements in the animation generation. Since many algorithms are update based, it might be desirable to have a specific syntax for update based graph descriptions. Another challenge is the improvement of the smoothness in animations by the removal of rough effects that appear when connected nodes move in different directions.

Nevertheless, using the prototype, further algorithms can be implemented and I hope that this thesis motivates researchers to improve, design, implement, or just to use algorithms for evolving graphs or trees to create useful and beautiful animations.

Bibliography

[1] Georgy. M. Adelson-Velsky and Evgenii. M. Landis. An Algorithm for the Organization of Information. *Doklady Akademii Nauk USSR*, 3(2):1259–1263, 1962.

[2] Dustin Lockhart Arendt and Leslie M. Blaha. SVEN: informative visual representation of complex dynamic structure. *CoRR*, abs/1412.6706, 2014.

[3] Benjamin Bach, Emmanuel Pietriga, and Jean-Daniel Fekete. Visualizing dynamic networks with matrix cubes. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 877–886. ACM Press, 2014.

[4] Fabian Beck, Michael Burch, Stephan Diehl, and Daniel Weiskopf. The state of the art in visualizing dynamic graphs. In *State of the Art Reports of the 16th Eurographics Conference on Visualization, EuroVis 2014*, pages 83–103. Eurographics Association, 2014.

[5] Paul Bonsma and Daniel Lokshtanov. Feedback vertex set in mixed graphs. In *Proceedings of the 12th International Conference on Algorithms and Data Structures, WADS 2011*, volume 6844 of *Lecture Notes in Computer Science*, pages 122–133. Springer-Verlag, 2011.

[6] Anne Brüggemann-Klein and Derick Wood. Drawing trees nicely with TEX. *Electronic Publishing*, 2(2):101–115, 1988.

[7] Ida Dorothee Bruhns. Design and implementation of a configureable framework for force-based graph drawing. Masterthesis, Institute of Theoretical Computer Science, Universität zu Lübeck, Lübeck, Germany, February 2014.

[8] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. Improving walker's algorithm to run in linear time. In *Proceedings of the 10th International Symposium on Graph Drawing, GD 2002*, volume 2528 of *Lecture Notes in Computer Science*, pages 344–353. Springer-Verlag, 2002.

[9] Michael Burch, Fabian Beck, and Daniel Weiskopf. Radial edge splatting for visualizing dynamic directed graphs. In *Proceedings of the International Conference*

on Computer Graphics Theory and Applications, IVAPP 2012, pages 603–612. SciTe Press, 2012.

[10] Michael Burch and Stephan Diehl. TimeRadarTrees: Visualizing dynamic compound digraphs. *Computer Graphics Forum*, 27(3):823–830, 2008.

[11] Robert F. Cohen, Giuseppe Di Battista, Roberto Tamassia, and Ioannis G. Tollis. Dynamic graph drawings: Trees, series-parallel digraphs, and planar st-digraphs. *SIAM Journal on Computing*, 24(5):970–1001, 1995.

[12] Robert F. Cohen, Giuseppe Di Battista, Roberto Tamassia, Ioannis G. Tollis, and Paola Bertolazzi. A framework for dynamic graph drawing. In *Proceedings of the 8th Annual Symposium on Computational Geometry, SCG 1992*, pages 261–270. ACM Press, 1992.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[14] Stephan Diehl and Carsten Görg. Graphs, they are changing. In *Proceedings of the 10th International Symposium on Graph Drawing, GD 2002*, volume 2528 of *Lecture Notes in Computer Science*, pages 23–30. Springer-Verlag, 2002.

[15] Stephan Diehl, Carsten Görg, and Andreas Kerren. Foresighted graphlayout. Technical report A/02/2000, FB Informatik, University Saarbrücken, Saarbrücken, Germany, December 2000.

[16] Stephan Diehl, Carsten Görg, and Andreas Kerren. Preserving the mental map using foresighted layout. In *Proceedings of the 3rd Joint Eurographics–IEEE TCVG Conference on Visualization*, volume 1, pages 175–184. The Eurographics Association, 2001.

[17] Paolo Federico, Wolfgang Aigner, Silvia Miksch, Florian Windhager, and Lukas Zenk. A visual analytics approach to dynamic social networks. In *Proceedings of the 11th International Conference on Knowledge Management and Knowledge Technologies*, i-KNOW '11, pages 47:1–47:8. ACM Press, 2011.

[18] Carsten Friedrich and Peter Eades. Graph drawing in motion. *Journal of Graph Algorithms and Applications*, 6(3):353–370, 2002.

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

[20] Marco Gaertler and Dorothea Wagner. A hybrid model for drawing dynamic and evolving graphs. In *Proceedings of the 13th International Symposium Graph Drawing, GD 2005*, volume 3843 of *Lecture Notes in Computer Science*, pages 189–200. Springer-Verlag, 2005.

[21] Carsten Görg, Peter Birke, Mathias Pohl, and Stephan Diehl. Dynamic graph drawing of sequences of orthogonal and hierarchical graphs. In *Proceedings of the 12th International Symposium on Graph Drawing, GD 2004*, volume 3383 of *Lecture Notes in Computer Science*, pages 228–238. Springer-Verlag, 2004.

[22] Martin Greilich, Michael Burch, and Stephan Diehl. Visualizing the evolution of compound digraphs with timearctrees. *Computer Graphics Forum*, 28(3):975–982, 2009.

[23] Georg Groh, Holger Hanstein, and Wolfgang Wörndl. Interactively visualizing dynamic social networks with DySoN. In *Proceedings of the Workshop on Visual Interfaces to the Social and the Semantic Web, VISSW2009*, February 2009.

[24] Petter Holme. Modern temporal network theory: a colloquium. *The European Physical Journal B*, 88(9):234:1–30, 2015.

[25] B. Johnson and B. Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2nd IEEE Conference on Visualization '91*, VIS '91, pages 284–291. IEEE Computer Society Press, October 1991.

[26] Richard Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[27] Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1(1):14–25, March 1971.

[28] Othon Michail. An introduction to temporal graphs: An algorithmic perspective. In *Algorithms, Probability, Networks, and Games - Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occasion of His 60th Birthday*, volume 9295 of *Lecture Notes in Computer Science*, pages 308–343. Springer-Verlag, 2015.

[29] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[30] Sven Moen. Drawing dynamic trees. *Software, IEEE*, 7(4):21–28, July 1990.

[31] Stephen C. North. Incremental layout in DynaDAG. In *Proceedings of the Symposium on Graph Drawing, GD '95*, volume 1027 of *Lecture Notes in Computer Science*, pages 409–418. Springer-Verlag, 1995.

[32] Stephen C. North and Gordon Woodhull. Online hierarchical graph drawing. In *Proceedings of the 9th International Symposium on Graph Drawing, GD 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 232–246. Springer-Verlag, 2001.

[33] Khairi Reda, Chayant Tantipathananandh, Andrew Johnson, Jason Leigh, and Tanya Berger-Wolf. Visualizing the evolution of community structures in dynamic social networks. *Computer Graphics Forum*, 30(3):1061–1070, 2011.

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

[35] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: Animated 3d visualizations of hierarchical information. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 189–194. ACM Press, 1991.

[36] John Stasko and Eugene Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *Proceedings of the IEEE Symposium on Information Vizualization 2000*, INFOVIS '00, pages 57–65. IEEE Computer Society, 2000.

[37] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[38] Kenneth J. Supowit and Edward M. Reingold. The complexity of drawing trees nicely. *Acta Informatica*, 18:377–392, 1982.

[39] Richard E. Sweet. Empirical estimates of program entropy. Report Stan-CS-78-698, Department of Computer Science, Stanford University, Stanford, CA, USA, November 1978.

[40] Till Tantau. *TikZ & PGF Manual for version 3.0.1a*, 2015. Available online at http://mirrors.ctan.org/graphics/pgf/base/doc/pgfmanual.pdf. Accessed April 2016.

[41] Till Tantau. Graph drawing in TikZ. In *Proceedings of the 20th International Symposium on Graph Drawing, GD 2012*, volume 7704 of *Lecture Notes in Computer Science*, pages 517–528. Springer-Verlag, 2012.

[42] John Q. Walker, II. A node-positioning algorithm for general trees. *Software: Practice and Experience*, 20(7):685–705, July 1990.

[43] Charles Wetherell and Alfred Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, 5(5):514–520, September 1979.

[44] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proc. VLDB Endow.*, 7(9):721–732, May 2014.