GoTree: A Grammar of Tree Visualizations

Guozheng Li[1], **Min Tian**[1], **Qinmei Xu**[1], **Michael J. McGuffin**[3], **Xiaoru Yuan**[1,2*]
[1]Key Laboratory of Machine Perception (Ministry of Education), Peking University
[2]National Engineering Laboratory for Big Data Analysis and Application, Peking University
[3]École de technologie supérieure
{guozheng.li, tianmin, mayqin_xu, xiaoru.yuan}@pku.edu.cn, michael.mcguffin@etsmtl.ca

ABSTRACT
We present GoTree, a declarative grammar allowing users to instantiate tree visualizations by specifying three aspects: visual elements, layout, and coordinate system. Within the set of all possible tree visualization techniques, we identify a subset of techniques that are both "unit-decomposable" and "axis-decomposable" (terms we define). For tree visualizations within this subset, GoTree gives the user flexible and fine-grained control over the parameters of the techniques, supporting both explicit and implicit tree visualizations. We developed Tree Illustrator, an interactive authoring tool based on GoTree grammar. Tree Illustrator allows users to create a considerable number of tree visualizations, including not only existing techniques but also undiscovered and hybrid visualizations. We demonstrate the expressiveness and generative power of GoTree with a gallery of examples and conduct a qualitative study to validate the usability of Tree Illustrator.

Author Keywords
Tree visualization; Declarative grammar; Authoring tool; Hierarchical data visualization.

CCS Concepts
•**Human-centered computing** → **Visualization toolkits; Visualization techniques;** *Information visualization;*

INTRODUCTION
Many techniques are available for visualizing hierarchical tree data, with the most extensive survey [50] covering over 300 techniques. A designer's choice of technique may depend on several factors, including the size and depth of the tree, the number of children per node, the number and types of attributes to be encoded (including text labels), space efficiency [38], legibility concerns (*e.g.*, should all text labels have the same orientation or the same size?), whether a layout is familiar to users, how the user can zoom or navigate within the tree, and whether depths of different nodes should be easy to compare [5, 32]. Furthermore, the most appropriate technique to use may change during a single user session (as tasks change) or from one subtree to another of the same tree (as a function of local characteristics).

To implement different layout techniques, a first strategy is to use a software library. One of the most popular is D3 [9], which supports several layout algorithms and can be extended with third-party libraries. However, libraries impose an upfront cost for the designer to learn an API, in addition to learning a programming language if it is not already known. Although D3 is widely used, this cost for beginners is nontrivial. Furthermore, achieving the flexibility necessary to investigate a larger set of layouts requires the programmer to write new low-level codes, incurring a substantial time investment for each new layout. Programmers will often instead resort to whatever layouts are provided by the available libraries. A second strategy is to use a declarative language, such as Vega [46]. By decoupling the specification from the execution, declarative grammars allow users to specify *what* to show without specifying *how* to render it. Vega supports tree layouts by naming an algorithm (*e.g.*, "slicedice"). Hence, users may only vary the parameters provided by these pre-defined tree visualizations without fine-grained control. A third strategy is to use the layouts built into end-user programs such as Tableau [1], which similarly limit the user to pre-defined techniques.

We present a new, simple method for non-expert users to design and construct desired tree visualizations more easily, which covers *a broad set* of techniques, but uses a small number of parameters that can be *incrementally modified* to enable the creative exploration of the set of possible techniques. This method could be used as a building block of higher-level tooling for research, education, and prototyping.

Some previous work has identified design dimensions for tree visualizations [50, 33, 18]. However, the dimensions are not sufficiently detailed to fully specify and instantiate a layout just from design choices. Other work *has* defined a more detailed set of independent design dimensions, with an algorithm, so that a set of choices fully specifies a layout [57, 6, 53]. However, these works were limited to implicit layouts, where parent-child relationships between nodes are shown by relative positioning.

The generative approach of Schulz et al. [51] can be thought of as a meta-algorithm. Rather than making design choices about the final output, a user instead specifies the operators to use at each stage of the layout process. This approach covers a larger set of possible layouts, but it is not obvious for a user

*indicates the corresponding author.

CHI '20, April 25–30, 2020, Honolulu, HI, USA.

ACM ISBN 978-1-4503-6708-0/20/04 ...$15.00.
http://dx.doi.org/10.1145/3313831.3376297

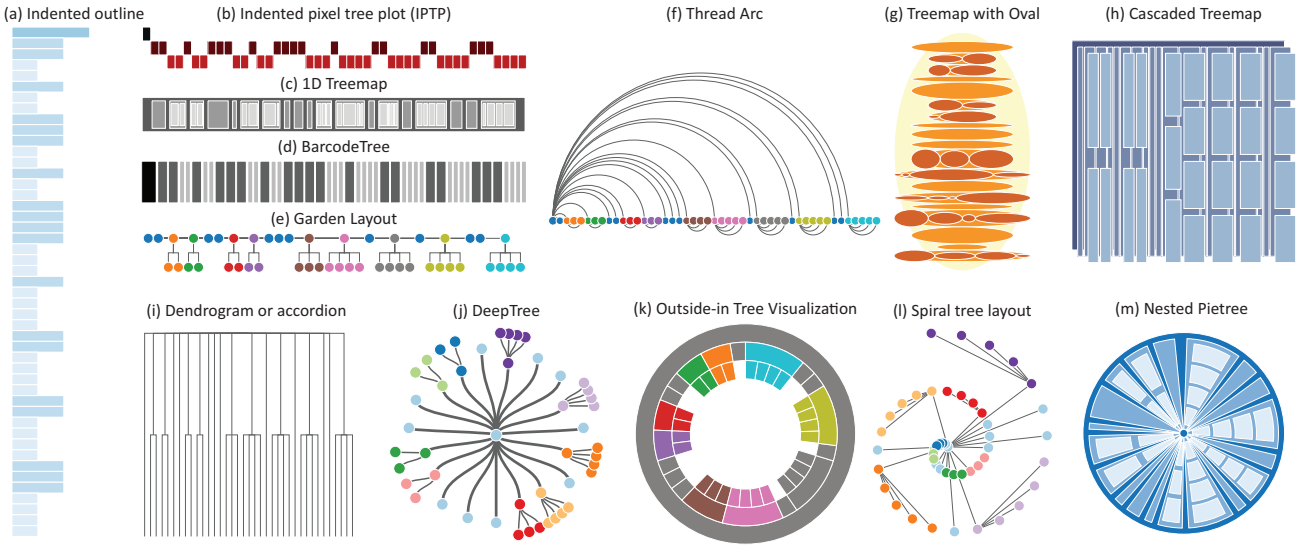

Figure 1. A gallery of 13 visualization examples generated with Tree Illustrator showing the expressiveness of GoTree. These tree visualizations are all unit-decomposable and axis-decomposable. (a) Indented outline [31], (b) Indented pixel tree plot (IPTP) [11], (c) 1D Treemap [40], (d) BarcodeTree [32], (e) Similar to Garden Tree Layout [15], (f) ThreadArc [29], (g) Similar to Ellimaps [41], (h) Cascaded Treemap [35], (i) Dendrogram or accordion [39], (j) DeepTree [7], (k) Outside-in tree visualization [28], (l) Similar to Spiral tree layout [16], (m) Nested Pietree [51]. More examples are at http://go-tree.info/gallery.html.

to know how to modify the instructions inside the operators to explore the set of possible layouts incrementally.

We present a declarative grammar, GoTree (**G**rammar **o**f **Tree** visualizations), for specifying tree visualizations that cover a large set of possible layouts (both explicit and implicit) *and* make it easy for a user to explore this set of layouts through incremental changes to a simple textual definition. GoTree allows users to instantiate tree visualizations by specifying three aspects: visual elements, layout, and coordinate system. For techniques that are "unit-decomposable" and "axis-decomposable" (terms we define below), GoTree decomposes the layout along each axis and gives users flexible, fine-grained control. We then transform the specifications into mathematical layout constraints and compute the resulting visualization using a constraint solver.

We implemented Tree Illustrator, an interactive tree visualization authoring tool based on GoTree. Tree visualization designs can be exported as images or reusable components in GoTree's JavaScript Object Notation (JSON) format, which can be used to visualize other hierarchical data.

We validate our work in two ways. First, we demonstrate GoTree's expressiveness with a gallery of examples (Figure 1 and our website), including previously undiscovered and hybrid techniques. Secondly, we validate the usability of Tree Illustrator through a usability test. Results show that Tree Illustrator allows users without a programming background to create tree visualizations efficiently.

Our contributions are (1) GoTree, a declarative grammar that covers a wide range of tree visualizations in a flexible and fine-grained manner; (2) Tree Illustrator, an interactive authoring tool based on GoTree, allowing users to create and explore tree visualizations, that was evaluated in a usability test.

RELATED WORK

Given the availability of previous surveys [50, 53], we only briefly introduce some conventional techniques and mainly discuss existing tree visualization frameworks. We then review related work on visualization grammars.

Tree Visualization

Tree visualizations are divided into explicit and implicit techniques according to the visual representations of parent-child relations. Explicit techniques show relationships as (polygonal) line segments or curves, such as in a hierarchical clustering tree. Implicit techniques show the same relationships using either adjacency, such as in icicle plots [31], or inclusion, such as in Treemaps [56]. Hybrid techniques [62] mix two or more approaches to combine their advantages.

In response to the many tree visualization techniques available, researchers have attempted to understand their design space. Treevis.net [50] classifies over 300 techniques along three dimensions: edge representation (explicit, implicit, hybrid), dimensionality (2D, 3D, hybrid), and node alignment (radial, axis-parallel, free). Li et al. [33] propose 12 design dimensions based on the items in treevis.net to understand tree visualizations from an evolutionary perspective. These works classify tree visualizations without seeking to generate them. Methods for generating visualizations based on a design-space require a layout procedure that takes design decisions as input, but the design dimensions of the above works are either not fine-grained enough or not independent, and inappropriate for an automatic procedure.

While it is difficult to capture all possible design choices with design-space-based methods, it is still possible for subclasses

of all possible tree visualizations. Schulz et al. [53] focused on implicit tree visualizations, which do not include node-link diagrams or other techniques with explicitly drawn edges. Their work divides the design space of implicit tree visualizations along four dimensions: dimensionality, node representation, edge representation, and layout. By restricting the scope to implicit techniques, they were able to derive an automatic procedure based on these design dimensions. To reduce the complexity of the creative procedure exposed to users, Schulz et al. [52] proposed a preset-based method, which allows users to specify tree visualizations by blending several existing visual representations (a.k.a., presets) instead of individual design choices. The above design-space-based methods either cannot instantiate a technique from a set of design choices or only support the creation of implicit or pre-defined tree visualizations.

In contrast, operator-based generative layout focuses on the algorithmic aspects of generating tree drawings. Some of the previous works of this type are limited to implicit tree visualizations, including work that uses operators to configure a hierarchical layout to show aspects of multivariate data [57, 36]. Baudel and Broeksema [6] use five dimensions (order, size, chunk, recurse, phrase) to drive space-filling layouts. Schulz et al. [51] propose an operator-based generative layout approach for both implicit and explicit tree drawings. Operators are placed in a pipeline with six stages: initialization, traversal, preprocess, prelayout, allocation, and postlayout. Their approach enables a wider range of tree visualizations, but at the cost of requiring designers to translate their intended visual design into an algorithmic description.

We compare GoTree with three declarative tree visualization authoring techniques [53, 51, 6] according to three criteria.

Abstraction level: GoTree affords users fine-grain control by decomposing layouts into relationships between components along each axis. Other works encapsulate algorithms in operators (e.g., Spiral [6], SLICE [51], subdivision [53]). Hence, in those other works, new layouts can require new operators, increasing programming effort and hindering extensibility.

Capability: The capabilities of GoTree and existing works [53, 51, 6] do not have simple superset/subset relations. Previous work cannot support certain tree visualizations (e.g., IPTP [11]) supported by GoTree. However, they also can support some tree visualizations (e.g., Squarified Treemap [37]), which cannot be described by GoTree, because they hide algorithm complexity in operators. The scope of GoTree covers unit-decomposable and axis-decomposable tree layouts, and its parameters are designed to cover a rich combinatorial set within that scope rather than just currently known tree visualizations. Our gallery demonstrates a range of capabilities. In contrast, some existing works [53, 6] are restricted to implicit or rectangular space-filling layouts, and their granularity [53, 6, 51] is limited by the use of pre-defined operators.

Construction difficulty: Existing works [6, 53] abstract the layout procedures into pipelines, whereas the components of GoTree can be represented graphically (Figure 3 and 4) and modified through a direct manipulation GUI (Tree Illustrator). Our work is more consistent with the mental model because users do not need to transform the desired tree visualizations into layout procedures mentally.

Grammars for Visualizations

We distinguish two ways to define a visualization programmatically. The first is to use imperative languages and libraries, which support the construction of visualizations from the beginning, including Prefuse [20], D3 [9], Processing [43], and Protovis [8]. By exposing the construction pipelines to developers, imperative programming provides great expressiveness at the cost of complexity and involves a steep learning curve for users. Imperative programming requires developers to focus more on implementation details and less on the visual representation. The second way is to use a declarative language, which is also widely used by the visualization community [47, 48, 49]. By decoupling the specification from the execution, declarative visualization grammars allow users to specify what is shown in visualizations directly without considering how the tree visualization is achieved [19].

Currently, many declarative visualization grammars exist with varying degrees of expressiveness. "The Grammar of Graphics" (GoG) [61] is one of the first declarative frameworks for visualization. Building upon GoG, Wickham implemented ggplot2 [60], a widely used R package for visualizations. Drawn from GoG and toolkits, including Protovis [8] and D3 [9], Vega [46] provides basic abstractions for constructing visualizations and extends the specification to interactive visualizations. Furthermore, Reactive Vega [48] provides a more comprehensive treatment of interaction design for data visualization based on event-driven reactive functional programming. To reduce the complexity of declarative grammar, Satyanarayan et al. [47] proposed Vega-Lite, a high-level declarative grammar that enables the rapid specification of interactive visualizations while reducing its expressiveness. Note that Vega-Lite cannot support the authoring of tree visualizations at the time of this writing.

Besides general-purpose declarative grammars, researchers have developed declarative approaches to provide fine-grained control for specific visualization categories or algorithms used in visualizations. In addition to the declarative grammars for tree visualizations introduced in the previous subsection, Park et al. [42] developed ATOM for unit visualizations. ATOM divides the space and data recursively until the size and position of every data item are determined. Declarative grammars also exist for multiclass density maps [26], volume visualizations [13, 55], and high-level constraints [22].

To our knowledge, a declarative grammar covering a wide range of tree visualizations does not exist in previous literature.

TREE VISUALIZATION CLASSIFICATION

Many tree visualization techniques can be naturally implemented as recursive algorithms where the region of space assigned to a node or subtree is computed only from local information (*i.e.*, computed from the parent, siblings, children,

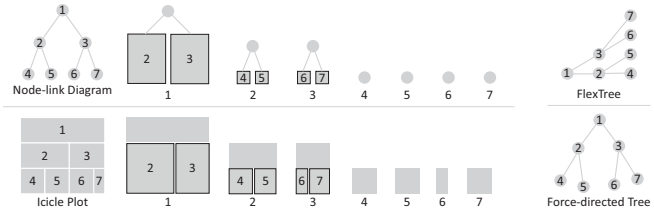

Figure 2. (a) Examples of unit-decomposable tree visualizations: classical node-link diagram, icicle plot, and the decomposed TreeUnits. (b) Examples that are not unit-decomposable: FlexTree, force-directed tree visualization.

and their assigned regions). This is true for many top-down and bottom-up recursive approaches.

We call such techniques *unit-decomposable*. The *units* that will be of interest for explaining GoTree are **TreeUnits**. Each TreeUnit consists of a node N and the subtrees under N's children. Node N can be thought of as a (local) **root node**, *i.e.*, the root node of the TreeUnit. The set of subtrees under N's children we refer to as the **subtree group** G, and we will use S to refer to an individual subtree of a child of N. In our work, the region assigned to each of these components of the TreeUnit will always be an axis-aligned rectangle (or, in the case of polar coordinates, an annulus sector).

Unit-decomposable techniques include the classical node-link diagram and icicle diagram [31] shown in Figure 2(a). Examples of non-unit-decomposable techniques include FlexTree [24] and force-directed tree visualization [14] (Figure 2(b)) because the layout of each node is determined by more than just local neighborhood.

To layout each TreeUnit, we need to determine the size (width, height) and position (x, y) of the unit and its contents. Within the category of unit-decomposable techniques, we further distinguish the subset of *axis-decomposable* techniques, where the assignment of positions and regions in the TreeUnit is done independently along each axis. Axis-decomposable techniques include slice-and-dice treemaps and icicle diagrams, whereas the following techniques are *unit*-decomposable but *not* axis-decomposable: circle packing [58], rectangular tesselation [2], squarified treemaps [10], and bubble treemap [17].

With unit-decomposable, axis-decomposable techniques, we observed that many techniques could be specified using two kinds of geometric relationships (Figure 3); this paper focuses on these tree visualization techniques.

TREEUNIT SPECIFICATION

With GoTree, a TreeUnit is defined in terms of the coordinate system, visual elements, and layout.

Coordinate System

The coordinate system determines the drawing space of tree visualizations with two parameters: dimensionality (2D or 3D) and category (Cartesian or Polar). The dimensionality and category parameters also influence two other aspects: visual elements and the layout. For example, rectangles in 2D

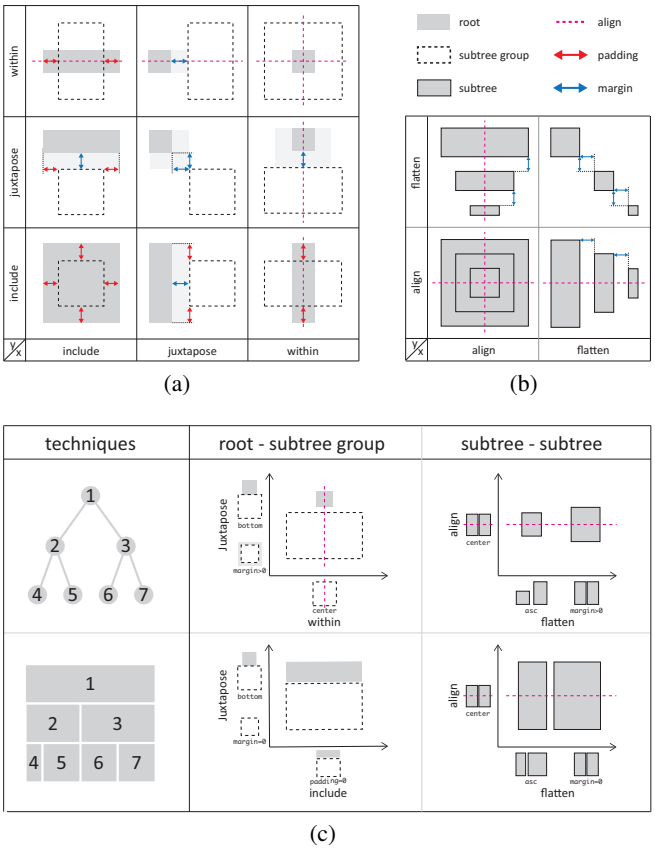

Figure 3. Relationships within one TreeUnit. (a) Relationships between the TreeUnit's root and subtree group. (b) Relationships among the subtrees within the subtree group. (c) Examples for node-link diagram and icicle plot.

cartesian space will change to annulus sectors in 2D polar space. The layout in 2D cartesian space requires users to specify the relationships along the x and y axes, but 2D polar space requires specification along the *angular* and *radial* axes. In the polar coordinate system, users can further customize tree visualizations by specifying the inner radius, central angle, start angle, and direction of the coordinate system. Note that the following specifications for mark and layout are assumed to be in 2D cartesian space. The tree visualization results will change with the categories of the coordinate system accordingly.

Visual Elements

Tree visualizations usually consist of two types of visual elements: nodes and links. The visual elements sometimes can be hidden; for example, the dendrogram [39] and DeepTree [7] do not show nodes, and implicit tree visualizations do not show links. Visual elements are determined by both shapes and visual styles. Based on the existing surveys of tree visualizations [50, 53] and further investigation, GoTree provides users with several shapes for nodes and links as well as the encoding approach for these visual elements. The shape of nodes could be a circle, rectangle, triangle, or ellipse. The shape of links could be a straight line, curved line, arc line, *etc*. To further determine the visualization results, users need

to specify the visual attributes based on selected shapes, including width, height, color, thickness. These visual styles can be either static or encoded with the attributes of hierarchical data (*e.g.*, depth, value, height).

Layout

GoTree allows users to define the geometric relationships between the TreeUnit's root and subtree group (Figure 3(a)), as well as between subtrees within the subtree group (Figure 3(b)) along each axis. There are three kinds of relationships between root and subtree group (*include*, *juxtapose*, *within*), and two kinds of relationships between the subtrees within the subtree group (*align*, *flatten*). Note that all the relationships are specified along each axis separately.

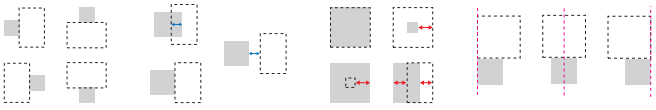

Figure 4. Parameters to specify geometry. (a) *position* parameter (left, right, top, bottom); (b) *margin* parameter (*margin* < *0*, *margin* = *0*, *margin* > *0*); (c) *padding* parameter (*padding* = *0*, *padding* < *0*, *padding* > *0*, *paddingLeft* > *0* and *paddingRight* < *0*); (d) *alignment* parameters (left, middle, right).

However, the above relationship specifications are insufficient to determine position and size. For example, the *within* parameter along the *x*-axis only determines that the horizontal geometric region of the root is placed inside the subtree group. However, the position of the root can be left, middle, or right relative to the subtree group. Therefore, we introduce more parameters inspired by CSS, which describe how the elements of web pages are displayed by graphical browsers. As shown in Figure 4, the parameters include padding, margin, position, and alignment. Together with these parameters, the layout specifications can capture all possible Allen relations [3] for intervals. Figure 5 introduces the margin and padding parameters along the *x*-axis. The values of these two parameters are relative to the whole TreeUnit or root node.

Figure 5. The padding and margin parameters along the *x*-axis.

TreeUnit Specification Framework

Figure 6 shows the declarative language framework of one TreeUnit, which is consistent with the TreeUnit decomposition above. The complete specification of GoTree can be found in the supplemental materials.

TREE VISUALIZATION SPECIFICATION

A TreeUnit only determines the relative positions between locally proximal components. GoTree allows TreeUnits to be assembled recursively, either using the same specification within each TreeUnit or using different specifications for different TreeUnits.

```
TreeUnitTemplate := CoordinateSystem, VisualElement, Layout

CoordinateSystem := Category, Dimensionality
Category := cartesian | polar    Dimensionality := 2d | 3d

VisualElement := Link, Node
Link := hidden | straight | orthogonal | arc | curve
Node := hidden | rectangle | triangle | circle | ellipse
Width, Height, Color, Thickness := static | depth | height
        | value | ...

Layout := (Root, Subtree){2..3}
Root-x, Root-y := within | juxtapose | include
Subtree-x, Subtree-y := flatten | align
within := alignment    include := padding
juxtapose := position, margin
align := alignment     flatten := sorting, margin

Notation   "|": or;    "{2..3}": two to three;
```

Figure 6. The formal specifications of one TreeUnit template in GoTree.

Tree Visualization Specification Framework

For unit-decomposable tree visualizations, TreeUnits are independent of nodes outside the TreeUnit. Therefore, users can specify the TreeUnits within one piece of hierarchical data as the same or different tree visualizations. As shown in Figure 8, one single tree visualization specification needs to determine three parts. The first part locates the root node of the target TreeUnits, the second part specifies whether to change all the descendant TreeUnits recursively, and the third part assigns one GoTree template for the selected TreeUnits.

In particular, for the TreeUnits repeatedly specified, more precise specifications have a higher priority. The priority of non-recursive specification is higher than the recursive one, and the id-based unit specification is higher than the property-based one, which includes the level, depth, name, and value. If two specifications have the same priority (*e.g.*, both of them are property-based specifications but different properties), the earlier one will have a higher priority.

```
TreeVisTemplate := TreeVisSpecifiction+
TreeVisSpecifiction := NodeQuery, Recursive, TreeUnitTemplate
NodeQuery := depth%2 == 1 | id == 'index1' | ...
Recursive := true | false

Notation   "+": one or more;    "|": or;
```

Figure 8. The formal specifications of one tree visualization template in GoTree.

Unit Assembly Approach

TreeUnits are assembled recursively with either a bottom-up or top-down traversal. This corresponds to assembling the TreeUnits in Figure 2(a), either right-to-left or left-to-right, respectively. If the user specifies a bottom-up traversal, the size of the subtrees in each TreeUnit is computed automatically (child TreeUnits become subtrees within their parent TreeUnit). However, with top-down traversal, the user may further specify additional options, to make the width and height of each TreeUnit's subtrees either "adaptive" (equal to each other), or computed from node depth, or an attribute value.

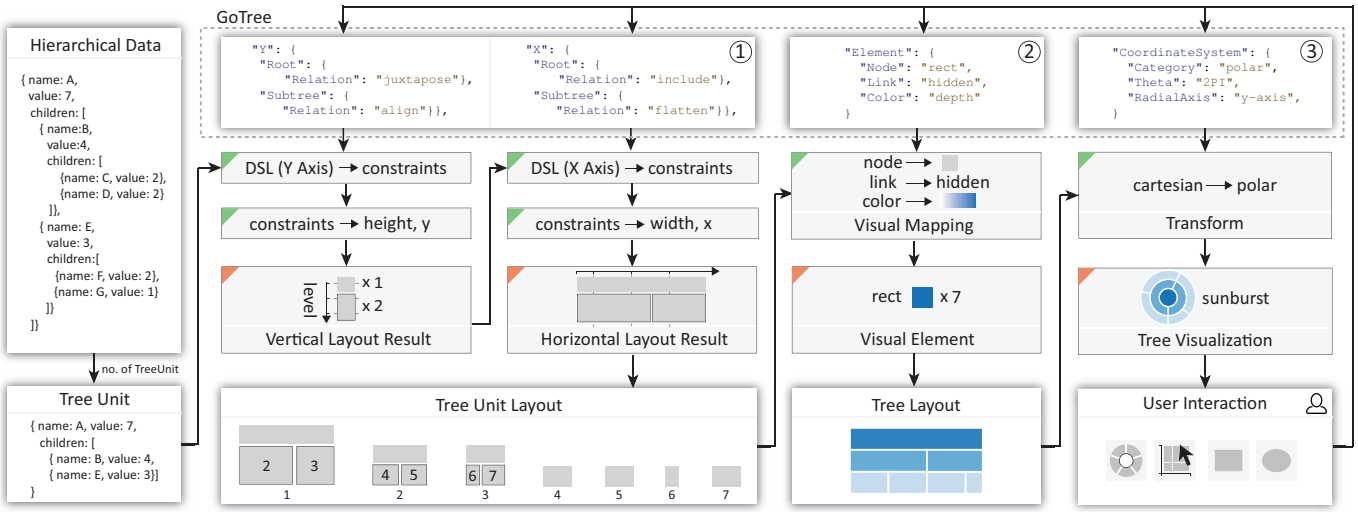

Figure 7. The computational pipeline of GoTree. The computations start from the original hierarchical data and then decompose into the TreeUnit array. By specifying the (1) *Layout*, (2) *Element*, and (3) *Coordinate system*, users will obtain the final tree visualizations. In particular, the computational procedures with green marks indicate the operations, and the computational procedures with orange marks indicate the intermediate results of tree visualization computation. After generating the tree visualizations, users can interact with the results to explore alternative tree visualizations.

TREE VISUALIZATION LAYOUT COMPUTATION

Our method calculates the tree visualization layouts by first parsing the GoTree specifications into constraints. For axis-decomposable tree visualizations, layout-related visual attributes (x, width, and y, height) are independent along each axis. Therefore, all the parsing results are linear constraints. Solving the linear constraints will enable the layout of specified tree visualizations to be obtained. The computational pipeline is shown in Figure 7.

GoTree Specification Parsing

A TreeUnit (T) contains one root node (N) and one subtree group (G) with several subtrees (S). For all these components, the shape of the occupied geometric regions is a rectangle in the cartesian coordinate system. Parsing the GoTree specifications is conducted to calculate the size (width, height) and positions (x, y) of these rectangular regions. We define the occupied regions of TreeUnit as R_T, the root node as R_N, subtree group as R_G, and the subtree as R_S.

In the following explanation, we only consider the specified relationships for the x-axis, as the possibilities for y-axis are the same. The specification for each axis consists of two parts: (1) the relationships between root node R_N and subtree group R_G, and (2) the relationships among subtree R_S within subtree group R_G.

Relations between root and subtree group

There are three kinds of relationships between root node R_N and subtree group R_G: *include*, *juxtapose*, and *within*. Each one implies certain constraints. *Within* means that R_N is inside R_G along an axis. In addition, we introduce the *alignment* parameter to specify the position of R_N. Figure 9 shows that R_N is at the center of R_G along the x-axis, and Equation 1 gives the corresponding constraints.

$$\begin{aligned} x(R_N) + \frac{width(R_N)}{2} &= \frac{width(R_T)}{2} \\ x(R_S) + \frac{width(R_S)}{2} &= \frac{width(R_T)}{2} \end{aligned} \tag{1}$$

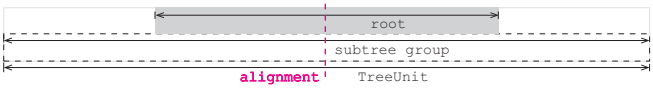

Figure 9. *Within* relationship between root and subtree group with *alignment* parameter set to *middle*.

Juxtapose means that R_N is adjacent to R_G along an axis. We introduce the *position* and *margin* parameters to specify their positions. Figure 10 shows that R_N is on the right side of R_G with distance $margin_{NG}$ along the x-axis, which is defined as the horizontal distance between R_N and R_G, and Equation 2 is the corresponding constraint.

$$width(R_N) + width(R_G) + margin_{NG} = width(R_T) \tag{2}$$

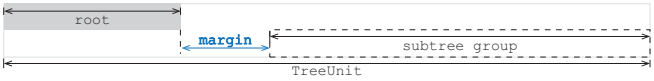

Figure 10. *Juxtapose* relationship between root and subtree group with setting *position* parameter to right.

Include means that R_N includes R_G along an axis. We introduce the *padding* parameter. Figure 11 shows that R_N includes R_G with distance *paddingLeft* and *paddingRight* along the x axis and Equation 3 gives the corresponding constraints.

$$\begin{aligned} width(R_N) &= width(R_T) \\ width(R_N) &= paddingLeft + paddingRight + width(R_S) \end{aligned} \tag{3}$$

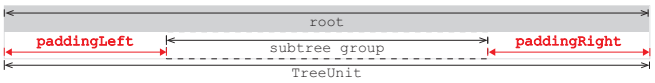

Figure 11. ***Include*** **relationship between root and subtree group with setting** ***paddingLeft*** **and** ***paddingRight*** **parameters.**

Relations among subtrees

There are two kinds of relationships between subtree R_S within subtree group R_G: *flatten* and *align*. *Flatten* indicates that each R_S within R_G has one separate space along one axis, and *margin* parameter controls their distance. Figure 12 shows that R_S is flattened within R_G with the *margin* distance set between them, and Equation 4 gives the corresponding constraints.

$$\begin{gathered} x(R_{S_1}) = x(R_G) \\ x(R_{S_i}) = x(R_{S_{i-1}}) + width(R_{S_{i-1}}) + margin_S, i = 2,\ldots,n-1 \\ x(R_{S_n}) + width(R_{S_n}) = x(R_G) + width(R_G) \end{gathered} \tag{4}$$

Figure 12. ***Flatten*** **relationship between subtrees with setting the** ***margin*** **parameter.**

Align parameter in the subtrees indicates that all R_S share the same space along one axis, and the *alignment* parameter controls their accurate positions. Figure 13 shows three subtrees aligned on the right side within R_G, and Equation 5 gives the corresponding constraints.

$$\begin{gathered} width(R_G) = \max_{i=1,\ldots,n} width(R_{S_{i-1}}) \\ \forall_{i=1,\ldots,n} x(R_{S_i}) + width(R_{S_i}) = x(R_G) + width(R_G) \end{gathered} \tag{5}$$

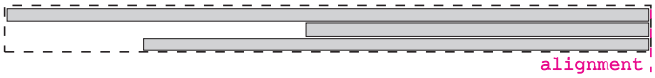

Figure 13. ***Alignment*** **relationship between subtrees within subtree group with right alignment.**

Constraint Solving

All the parsing results of axis-decomposable tree visualizations are linear constraints. Therefore, we use a linear solver, which is fast and guaranteed convergence, to meet the requirements of interactive parameter adjustment. For incomplete tree layout specifications, the linear constraint solver computes the optimal solution of layout attributes with the least-square method. Many existing algorithms [4, 54, 21] in the literature can solve linear constraints. Similar to Charticulator [45], the constraints of the GoTree layout specification involve only a few variables and produce a very sparse matrix. Therefore, we also adopted the Conjugate Gradient algorithm [54], which is efficient for solving sparse linear systems but only supports equality constraints. More details about the computational efficiency of solving constraints can be found in the supplementary material.

TREE ILLUSTRATOR

We have designed and implemented a prototype system called Tree Illustrator to support users in creating tree visualizations based on GoTree interactively.

Design Principle

Reduce the cognitive burden for constructing the tree visualizations based on GoTree. As a declarative language, GoTree decouples the specification (the "what") from execution (the "how") [19], which enables users to focus on visual encoding decisions instead of implementation details. Compared with the imperative programming approach, the specifications of GoTree do not require users to translate their intended visual design into functional aspects. Therefore, they are more consistent with the users' mental model. Users can write the GoTree JSON file to create tree visualizations directly. However, this still requires users to remember GoTree's parameters and convert the target tree visualizations into parameters cognitively. To reduce the cognitive burden, Tree Illustrator uses preview images in Figure 3(a) and (b) to represent corresponding parameters. With the preview images, users can choose them directly according to the target tree visualizations instead of typing in the parameters.

Balance direct manipulation and configuration widgets. The Direct manipulation interaction (*e.g.*, click to select and object with a draggable anchor) exists in many visualization authoring tools, including Data-driven Guides [30] and Charticulator [45]. After determining the relationships between components, Tree Illustrator supports the direct manipulation interaction for adjusting some parameters, including the margin and padding. However, for some parameters, the visualization results after direct manipulation cannot convert back for obtaining the parameters easily (*e.g.*, different relationships between components). Therefore, we design the configuration panels for these parameters to allow users to adjust the values directly. Furthermore, to help users associate the components and related parameters, the widgets will be highlighted when selecting the components.

User Interface and Interaction

The user interface of Tree Illustrator consists of Component, TreeUnit, Template, and Tree canvas panels (Figure 14).

Consistent with the design of GoTree, the component panel consists of three parts: the layout, visual elements, and coordinate system. Each component in the component panel uses one preview image to represent the underlying parameters. Users can select the component by clicking on the preview images, which is equivalent to writing the GoTree JSON file.

Users can create TreeUnits or adjust the parameters of an existing TreeUnit in the TreeUnit panel, which consists of two parts. The canvas panel above shows the visualization results of the simplest hierarchical data as well as the visual representations of the parameters (*e.g.*, the pink dashed line indicates the *alignment* parameter). The configuration panel below shows the parameter widgets of the selected components in this TreeUnit. Users can adjust the TreeUnit through direct

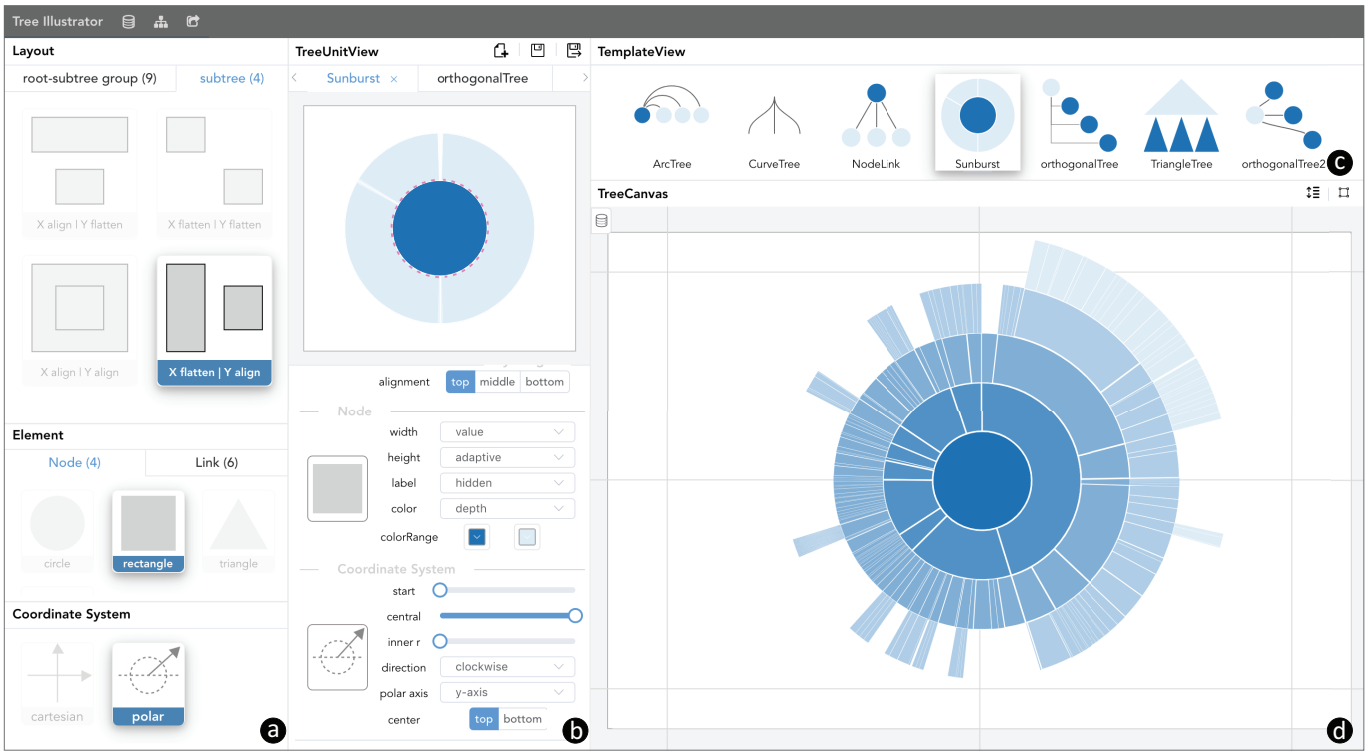

Figure 14. The interface of Tree Illustrator. (a) Tree component panel. (b) TreeUnit panel. (c) Tree visualization template panel. (d) Tree canvas panel.

manipulations in the canvas panel or parameter widgets in the configuration panel.

After finishing the TreeUnit design, users can save it in the Template panel, which contains many tree visualization templates. Each tree visualization template contains one preview image, and the underlying data of the preview image is the same as that of the TreeUnit panel. Clicking on one preview image in the Template panel will add the corresponding TreeUnit into the TreeUnit panel and visualize the selected hierarchical data in the Tree Canvas panel.

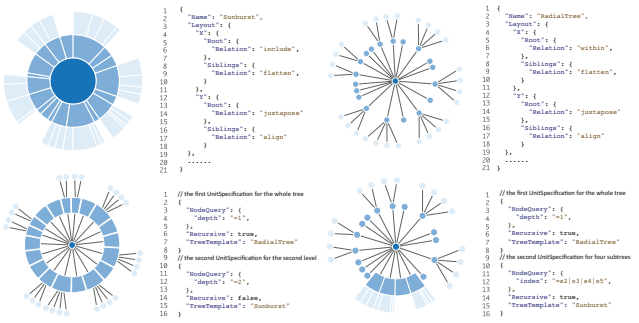

Figure 15. Top row: GoTree templates and visualization results for sunburst and radial tree layout. Bottom row: two hybrid tree visualizations, changing one level of the radial tree to sunburst (left) and changing one subtree of the radial tree to sunburst (right).

The tree visualization results for the selected or uploaded hierarchical data are shown in the Tree Canvas panel. In this panel, users can specify the GoTree template for each TreeUnit in the target hierarchical data and how to assemble the TreeUnits as one tree visualization (top-down or bottom-up). In particular, users can specify the TreeUnit as a different GoTree template to create hybrid tree visualizations.

Implementation

To leverage the capabilities of the existing visualization libraries, GoTree is implemented as an embedded declarative language [25] within JavaScript. The format of GoTree is based on JSON (JavaScript Object Notation), a widely used standard and supported in many programming languages. Furthermore, JSON is easy to parse and has sufficient expressiveness. Tree Illustrator is implemented as an HTML5 application and uses technologies, including Vue and NodeJS. The rendering part of Tree Illustrator utilizes D3 [9] based on Scalable Vector Graphics (SVG).

EVALUATION

In addition to our gallery of examples showing GoTree's expressive power, we also conducted a reproduction test to validate the usability of Tree Illustrator. Because the design of Tree Illustrator is consistent with, and flows directly from, GoTree, the evaluation also indirectly validates GoTree.

Visualization Gallery

We created a diverse tree visualization gallery using Tree Illustrator to demonstrate the expressiveness and generative power of GoTree. The complete gallery is available on our companion website, and includes previously known tree visualizations (*e.g.*, Figure 1) as well as hybrids (*e.g.*, Figure 15) and several novel visualizations. Of the novel visualizations,

one of the more interesting ones we named ClockTree (Figure 16), which maps nodes to circular sectors in a depth-first traversal order and arranges them counterclockwise. ClockTree does not show parent-child relations explicitly; users need to scan the nodes sequentially to discern the underlying topology. However, one benefit of ClockTree is that it can use the inner space for arcs to show a different set of relationships between pairs of nodes (Figure 16, right). This is somewhat like Figure 13b in Holten [23], with the advantage that ClockTree can show arcs between *any* pair of nodes rather than just between leaf nodes.

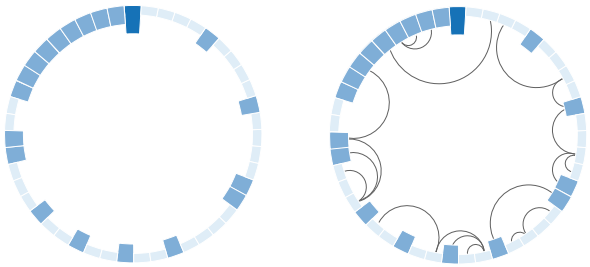

Figure 16. Novel ClockTree visualization created with Tree Illustrator. Color and node length (along the radial axis) encode node depth. Specifically, the dark-blue node is the root, the medium-blue nodes are children, the light-blue nodes are grandchildren, and nodes are arranged around the circle in depth-first order. Right subfigure: arcs can be added to show additional relationships between nodes.

Usability Study

To evaluate whether users can understand the design of GoTree and create tree visualizations using Tree Illustrator, we conducted a study to ask users to reproduce given tree visualizations. The procedures of the study follow the evaluation of Data Illustrator [34] and Charticulator [45].

Participants and Apparatus. 21 participants (7 female, 14 male) from five different departments in a university were divided into two groups according to their background. Those in the first group (2 female, 9 male) have a computer science (CS) background and have participated in the development of at least one project. Those in the second group (5 female, 5 male) are from liberal arts and social sciences departments, with no CS background. Tasks were performed in a quiet computer lab on a Dell Precision T5500 desktop PC, with an Intel Xeon Quad-Core processor, 8GB RAM, and an Nvidia Quadro 2000 graphics card driving two 23-inch LCD 1920 $\times$ 1080 pixel monitors, the left one showing the target tree visualizations, and the right one showing the user interface of Tree Illustrator.

Tasks. We prepared four tree visualization reproduction tasks: Indented pixel tree plot (Figure 1(b)) for Task1, Thread Arc (Figure 1(f)) for Task2, half sunburst with spacing between different levels (similar to Figure 14(d)) for Task3, and hybrid tree visualization of radial tree and sunburst (Figure 15) for Task4. These tasks cover all basic concepts of GoTree and main functionalities of Tree Illustrator: specifying different relations between the components and setting their parameters; adjusting styles of visual elements; changing parameters of the polar coordinate system; and constructing hybrid tree visualizations. The underlying hierarchical dataset we used in the study is the package structure of Flare[1].

Procedures. In the beginning, participants are asked to complete a pre-study background questionnaire. Then we provide a tutorial on GoTree. After the explanation, we asked the participants to practice decomposing node-link and icicle diagrams on paper. This part took around 25 minutes. After that, we began to introduce the functionalities and operations of Tree Illustrator. Then we introduced the example hierarchical dataset in the experiment and asked the participants to use Tree Illustrator to generate node-link and icicle diagrams based on previous decomposition results. This part took approximately 20 minutes. During these practice tasks, we encouraged participants to think aloud and ask questions.

After getting familiar with GoTree and Tree Illustrator, participants performed the four tasks described above on their own. Before each task, we describe the target chart for the participants. When participants are ready, they click the "start" button to begin the task. After finishing the target tree visualizations, participants click the "complete" button to finish the task, and the system will record the time cost automatically. Participants are encouraged to complete these four tasks independently and think aloud. We provide participants with instructions if they get stuck on the functionalities of Tree Illustrator or task descriptions. After completing four tasks, we invited the participants to play with our system freely, trying different options, and exploring different tree visualizations. Participants were asked to fill out a questionnaire about their experience of learning and using Tree Illustrator. At last, we interviewed each participant and collected their comments on both the GoTree and Tree Illustrator. The entire session lasted around 1.5 hours for each participant.

Results

All participants can reproduce the four target tree visualizations successfully with a few guidance. The guidance for the participants is mainly about the correspondence between visual effect and parameters. For example, *"where could I adjust the alignment parameters?"*(P10 and P14). To address this issue, we added interactive highlighting between the Parameter view and TreeUnit Canvas in the TreeUnit panel (Figure 14(b)). We asked the participants to try Tree Illustrator further and ask their feedbacks about the revision. All the participants completed the tasks without help and agreed that the interaction addresses this issue well.

Figure 17 shows the results of task completion time. We are interested in whether the participants' background influences their learning and experience of GoTree and Tree Illustrator. From Figure 17, we learned that the completion time of Task1 and Task2 are similar between Group1 and Group2. For Task3 and Task4, the completion time of Group2 is slightly larger than Group1. We interviewed the participants further and found that tree visualizations in the third and fourth tasks are built in a polar coordinate system. GoTree requires users

[1] https://github.com/d3/d3-hierarchy/blob/master/test/data/flare.json

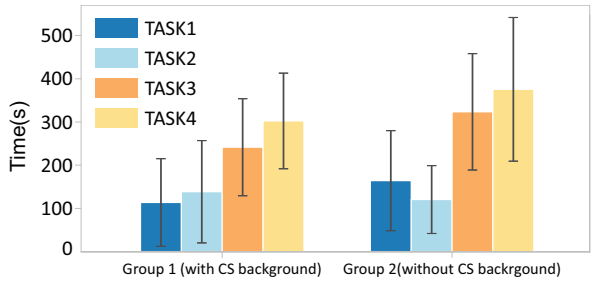

Figure 17. Task completion time of the participates with different backgrounds. Error bars indicate the standard deviations.

to specify the visualizations along the axis, but the specifications are designed in the cartesian coordinate system. It is challenging to transforming the relations in the cartesian system to the polar system, especially for the participants without a CS (Computer Science) background. Future work could extend the user interface to display circular sliders, icons, and previews when the polar system is selected.

Participants rated GoTree and Tree Illustrator on four satisfaction criteria using a five-point Likert scale. They indicate that GoTree and Tree Illustrator are easy to learn (GoTree: μ=4.09, σ=0.94; Tree Illustrator: μ=4.00, σ=1.02), Tree Illustrator is easy to operate (μ=4.18, σ=0.60), and enjoyable to create tree visualizations (μ=4.45, σ=0.52). During the interview, several participants commented on the expressiveness: *"I am impressive that [Tree Illustrator] can help me create so many different tree visualizations just using drag and drop."* (P3) and usability *"[GoTree] gives me many more options than D3.... the construction of the hybrid tree visualizations is great!"* (P6). Such comments are consistent with previous findings: The decoupling of declarative language lets users focus on visual encoding decisions instead of implementation details [49].

DISCUSSION AND FUTURE WORK

In contrast with GoTree, the competing solutions differ in the conceptual framework, expressiveness, availability of tutorial, and of the prototype system. Trying to compare them empirically would involve confounding variables — any differences found could be due to a mix of factors. Therefore, we did not conduct a comparative user experiment.

As a grammar of unit-decomposable and axis-decomposable tree visualizations, GoTree is consistent and complete. Regarding consistency, the computations of visual attributes along the axes of different TreeUnits are independent. Thus GoTree does not violate the basic design principle of unit-decomposable and axis-decomposable tree visualizations. GoTree can also prevent "over-specification" of constraints because users can only specify one parameter for each relationship along each axis. Regarding completeness, GoTree is designed to capture all the relative positions (between parent-child and among siblings) within the tree visualizations. The parameters are designed to cover all possible Allen relations [3] rather than covering the currently known tree visualizations. Our tree visualization gallery is also an indication of the design space's completeness.

Currently, 237 two-dimensional tree visualizations exist in the treevis.net gallery. Among these tree visualizations, GoTree can support around 100 techniques. The unsupported tree visualization techniques are mainly divided into four categories: (1) tree visualizations that use the tree or map visual metaphor, (2) techniques that combine the tree with other visualizations, (3) techniques that focus on the rendering (*e.g.*, shading) or interaction methods instead of the tree layout, and (4) techniques that improve the tree visualization results based on the existing tree visualization layout. For example, Squarified Treemap [10] improves the aspect ratio of nodes in the slice-dice-treemap [27], the Reingold-Tilford layout [44] improves the compactness of the tree layout proposed by Wetherell & Shannon [12].

The specification of GoTree does not only consider the interactions of tree visualizations. The created tree visualizations only provide some default interactions, including hovering on one node to show the information tooltip and clicking on one node to select it. Though important, we leave the specification of related interaction techniques (*e.g.*, collapsing or expanding) to future work.

GoTree only allows users to specify the visual elements as four basic shapes: circle, rectangle, triangle, and ellipse. To improve its expressiveness, Tree Illustrator will support users in creating more intuitive node element designs by drawing arbitrary polylines as data-driven guides [30] or uploading the image to create infographics with InfoNice [59].

GoTree exposes a large design space of tree visualizations. For future work, we would also like to explore undiscovered novel tree visualizations automatically based on GoTree. Recommending the most appropriate tree visualizations for users automatically according to the underlying hierarchical data and tasks is also a direction that warrants further investigation.

CONCLUSION

We present GoTree, a declarative grammar for creating a wide range of tree visualizations by specifying three aspects: visual elements, coordinate system, and layout. GoTree decomposes the layout further into two kinds of relationships (between the root and subtree group, among subtrees within the subtree group) along each axis. With decomposition, GoTree provides users flexible and fine-grained control for tree visualizations. Based on GoTree, we design and build Tree Illustrator, an interactive tree visualization authoring tool. We demonstrate the expressiveness of GoTree grammar through visualization examples. A reproduction study validates the usability of Tree Illustrator and shows that the system is learnable for users without a programming background. Tree Illustrator based on GoTree is available at `http://go-tree.info`.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. This work is supported by NSFC No. 61672055 and the National Key Research and Development Program of China (2016QY02D0304).

REFERENCES

[1] https://www.tableau.com/.

[2] Narendra Ahuja. 1986. Efficient planar embedding of trees for VLSI layouts. *Computer Vision, Graphics, and Image Processing* 34, 2 (1986), 189–203.

[3] James F. Allen. 1983. Maintaining Knowledge about Temporal Intervals. *Commun. ACM* 26, 11 (1983), 832–843.

[4] Greg J. Badros, Alan Borning, and Peter J. Stuckey. 2001. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Transactions on Computer-Human Interaction* 8, 4 (2001), 267–306.

[5] S. Todd Barlow and Padraic Neville. 2001. A comparison of 2-D visualizations of hierarchies. In *Proc. IEEE Symp. Information Visualization (InfoVis)*. 131–138.

[6] Thomas Baudel and Bertjan Broeksema. 2012. Capturing the Design Space of Sequential Space-Filling Layouts. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (2012), 2593–2602.

[7] Florian Block, Michael S Horn, Brenda Caldwell Phillips, Judy Diamond, E Margaret Evans, and Chia Shen. 2012. The DeepTree exhibit: Visualizing the tree of life to facilitate informal learning. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (2012), 2789–2798.

[8] Michael Bostock and Jeffrey Heer. 2009. Protovis: A Graphical Toolkit for Visualization. *IEEE Transactions on Visualization and Computer Graphics* 15, 6 (2009), 1121–1128.

[9] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (2011), 2301–2309.

[10] Mark Bruls, Kees Huizing, and Jarke J. van Wijk. 2000. Squarified Treemaps. In *Proc. Eurographics and IEEE TCVG Symposium on Visualization (VisSym)*. 33–42.

[11] Michael Burch, Michael Raschke, and Daniel Weiskopf. 2010. Indented Pixel Tree Plots. In *Proc. Int. Symp. Advances in Visual Computing (ISVC)*. 338–349.

[12] Wetherell Charles and Shannon Alfred. 1979. Tidy Drawings of Trees. *IEEE Transactions on Software Engineering* SE-5, 5 (1979), 514–520.

[13] Hyungsuk Choi, Woohyuk Choi, Tran Minh Quan, David G. C. Hildebrand, Hanspeter Pfister, and Won-Ki Jeong. 2014. Vivaldi: A Domain-Specific Language for Volume Processing and Visualization on Distributed Heterogeneous Systems. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2407–2416.

[14] Ana M. Cuadros, Fernando Vieira Paulovich, Rosane Minghim, and Guilherme P. Telles. 2007. Point Placement by Phylogenetic Trees and its Application to Visual Analysis of Document Collections. In *Proc. IEEE Symp. Visual Analytics Science And Technology (VAST)*. 99–106.

[15] Peter Eades. 1992. Drawing free trees. *Bulletin of the Institute of Combinatorics and its Applications* 5 (1992), 10–36.

[16] David Eppstein. 2009. Visualizing BFS as a spiral. (2009).

[17] Jochen Görtler, Christoph Schulz, Daniel Weiskopf, and Oliver Deussen. 2018. Bubble Treemaps for Uncertainty Visualization. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2018), 719–728.

[18] Wang Guanqun, Nakanishi Tsuneo, and Fukuda Akira. 2016. 2-D Layout for Tree Visualization: a survey. *MATEC Web of Conferences* 56 (2016), 01007.

[19] Jeffrey Heer and Michael Bostock. 2010. Declarative Language Design for Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics* 16, 6 (2010), 1149–1156.

[20] Jeffrey Heer, Stuart K. Card, and James A. Landay. 2005. Prefuse: a toolkit for interactive information visualization. In *Proc. ACM Conf. Human Factors in Computing Systems (CHI)*. 421–430.

[21] Allan Heydon and Greg Nelson. 1994. *The Juno-2 constraint-based drawing editor*. Technical Report 131a. Digital Systems Research.

[22] Jane Hoffswell, Alan Borning, and Jeffrey Heer. 2018. SetCoLa: High-Level Constraints for Graph Layout. *Computer Graphics Forum* 37, 3 (2018), 537–548.

[23] Danny Holten. 2006. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (2006), 741–748.

[24] Edwin P Curran Hongzhi Song and Roy Sterritt. 2002. FlexTree: visualising large quantities of hierarchical information. In *Proc. IEEE International Conference on Systems, Man and Cybernetics (SMC)*.

[25] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM Computing Surveys (CSUR)* 28, 4es (1996), 196.

[26] Jaemin Jo, Frédéric Vernier, Pierre Dragicevic, and Jean-Daniel Fekete. 2019. A Declarative Rendering Model for Multiclass Density Maps. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (2019), 470–480.

[27] Brian Johnson and Ben Shneiderman. 1991. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Proc. IEEE Visualization (VIS)*. 284–291.

[28] Thomas A. Keahey, Daniel J. Rope, and Graham J. Wills. 2018. Generating an Outside-in Hierarchical Tree Visualization. (2018).

[29] Bernard Kerr. 2003. Thread Arcs: an email thread visualization. In *Proc. IEEE Symp. Information Visualization (InfoVis)*. 211–218.

[30] Nam Wook Kim, Eston Schweickart, Zhicheng Liu, Mira Dontcheva, Wilmot Li, Jovan Popovic, and Hanspeter Pfister. 2016. Data-driven guides: Supporting expressive design for information graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2016), 491–500.

[31] Joseph B Kruskal and James M Landwehr. 1983. Icicle plots: Better displays for hierarchical clustering. *The American Statistician* 37, 2 (1983), 162–168.

[32] Guozheng Li, Yu Zhang, Yu Dong, Jie Liang, Jinson Zhang, Jinsong Wang, Michael J. McGuffin, and Xiaoru Yuan. 2020. BarcodeTree: Scalable Comparison of Multiple Hierarchies. *IEEE Transactions on Visualization and Computer Graphics* 26, 1 (2020), 1022–1032.

[33] Shaomeng Li, R Jordan Crouser, Garth Griffin, Connor Gramazio, Hans-Jörg Schulz, Hank Childs, and Remco Chang. 2015. Exploring hierarchical visualization designs using phylogenetic trees. In *Visualization and Data Analysis*. International Society for Optics and Photonics, SPIE, 68 – 81.

[34] Zhicheng Liu, John Thompson, Alan Wilson, Mira Dontcheva, James Delorey, Sam Grigg, Bernard Kerr, and John Stasko. 2018. Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring. In *Proc. ACM Conf. Human Factors in Computing Systems (CHI)*. ACM, 123:1–123:13.

[35] Hao Lü and James Fogarty. 2008. Cascaded treemaps: examining the visibility and stability of structure in treemaps. In *Proceedings of graphics interface*. 259–266.

[36] S MacNeil and Niklas Elmqvist. 2013. Visualization mosaics for multivariate visual exploration. *Computer Graphics Forum* 32, 6 (2013), 38–50.

[37] Jarke J. van Wijk Mark Bruls, Kees Huizing. 2000. Squarified Treemaps. In *Proc. Eurographics / IEEE VGTC Conference on Visualization (EuroVis)*. 33–42.

[38] Michael J. McGuffin and Jean-Marc Robert. 2010. Quantifying the Space-Efficiency of 2D Graphical Representations of Trees. *Information Visualization* 9, 2 (2010), 115–140.

[39] Tamara Munzner, François Guimbretière, Serdar Tasiran, Li Zhang, and Yunhong Zhou. 2003. TreeJuxtaposer: Scalable Tree Comparison Using Focus+Context with Guaranteed Visibility. *ACM Transactions on Graphics* 22, 3 (2003), 453–462.

[40] Petra Neumann, Stefan Schlechtweg, and Sheelagh Carpendale. 2005. ArcTrees: Visualizing Relations in Hierarchical Data. In *Proc. Eurographics / IEEE VGTC Conference on Visualization (EuroVis)*. 53–60.

[41] Benoît Otjacques, Monique Noirhomme, Xavier Gobert, Pierre Collin, and Fernand Feltz. 2007. Visualizing the activity of a web-based collaborative platform. In *Proc. Int. Conf. Information Visualisation (IV)*. IEEE Computer Society, 251–256.

[42] Deokgun Park, Steven M. Drucker, Roland Fernandez, and Niklas Elmqvist. 2018. Atom: A Grammar for Unit Visualizations. *IEEE Transactions on Visualization and Computer Graphics* 24, 12 (2018), 3032–3043.

[43] Casey Reas and Ben Fry. 2007. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press.

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

[45] Donghao Ren, Bongshin Lee, and Matthew Brehmer. 2019. Charticulator: Interactive Construction of Bespoke Chart Layouts. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (2019), 789–799.

[46] Arvind Satyanarayan and Jeffrey Heer. 2014. Lyra: An Interactive Visualization Design Environment. *Computer Graphics Forum* 33, 3 (2014), 351–360.

[47] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 341–350.

[48] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (2016), 659–668.

[49] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative Interaction Design for Data Visualization. In *Proc. ACM Conf. Symposium on User Interface Software and Technology (UIST)*. ACM, 669–678.

[50] Hans-Jörg Schulz. 2011. Treevis.net: A Tree Visualization Reference. *IEEE Computer Graphics and Applications* 31, 6 (2011), 11–15.

[51] Hans-Jörg Schulz, Zabedul Akbar, and Frank Maurer. 2013. A generative layout approach for rooted tree drawings. In *Proc. IEEE Pacific Visualization Symposium (PacificVis)*. 225–232.

[52] Hans-Jörg Schulz and Steffen Hadlak. 2015. Preset-based generation and exploration of visualization designs. *Journal of Visual Languages and Computing* 31 (2015), 9–29.

[53] Hans-Jörg Schulz, Steffen Hadlak, and Heidrun Schumann. 2011. The Design Space of Implicit Hierarchy Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics* 17, 4 (2011), 393–411.

[54] Jonathan Richard Shewchuk and others. 1994. An introduction to the conjugate gradient method without the agonizing pain. (1994).

[55] Min Shih, Charles Rozhon, and Kwan-Liu Ma. 2019. A Declarative Grammar of Flexible Volume Visualization Pipelines. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (2019), 1050–1059.

[56] Ben Shneiderman. 1992. Tree Visualization with Tree-maps: 2-d Space-filling Approach. *ACM Transactions on Graphics* 11, 1 (1992), 92–99.

[57] Aidan Slingsby, Jason Dykes, and Jo Wood. 2009. Configuring Hierarchical Layouts to Address Research Questions. *IEEE Transactions on Visualization and Computer Graphics* 15, 6 (2009), 977–984.

[58] Weixin Wang, Hui Wang, Guozhong Dai, and Hongan Wang. 2006. Visualization of Large Hierarchical Data by Circle Packing. In *Proc. ACM Conf. Human Factors in Computing Systems (CHI)*. ACM, 517–520.

[59] Yun Wang, Haidong Zhang, He Huang, Xi Chen, Qiufeng Yin, Zhitao Hou, Dongmei Zhang, Qiong Luo, and Huamin Qu. 2018. InfoNice: Easy Creation of Information Graphics. In *Proc. ACM Conf. Human Factors in Computing Systems (CHI)*. ACM, 335:1–335:12.

[60] Hadley Wickham. 2016. *ggplot2: Elegant Graphics for Data Analysis*. Springer.

[61] Leland Wilkinson. 2006. *The Grammar of Graphics*. Springer.

[62] Shengdong Zhao, Michael J. McGuffin, and Mark H. Chignell. 2005. Elastic Hierarchies: Combining Treemaps and Node-Link Diagrams. In *Proc. IEEE Symp. Information Visualization (InfoVis)*. 57–64.