Vision, Modeling, and Visualization (2020)
J. Krüger, M. Niessner, and J. Stückler (Eds.)

# Data Reconstruction from Colored Slice-and-Dice Treemaps

M. Henkel[1], V. Knauthe[1], T. von Landesberger[1], S. Guthe[1,2]

[1]TU Darmstadt, Germany
[2]Fraunhofer IGD, Germany

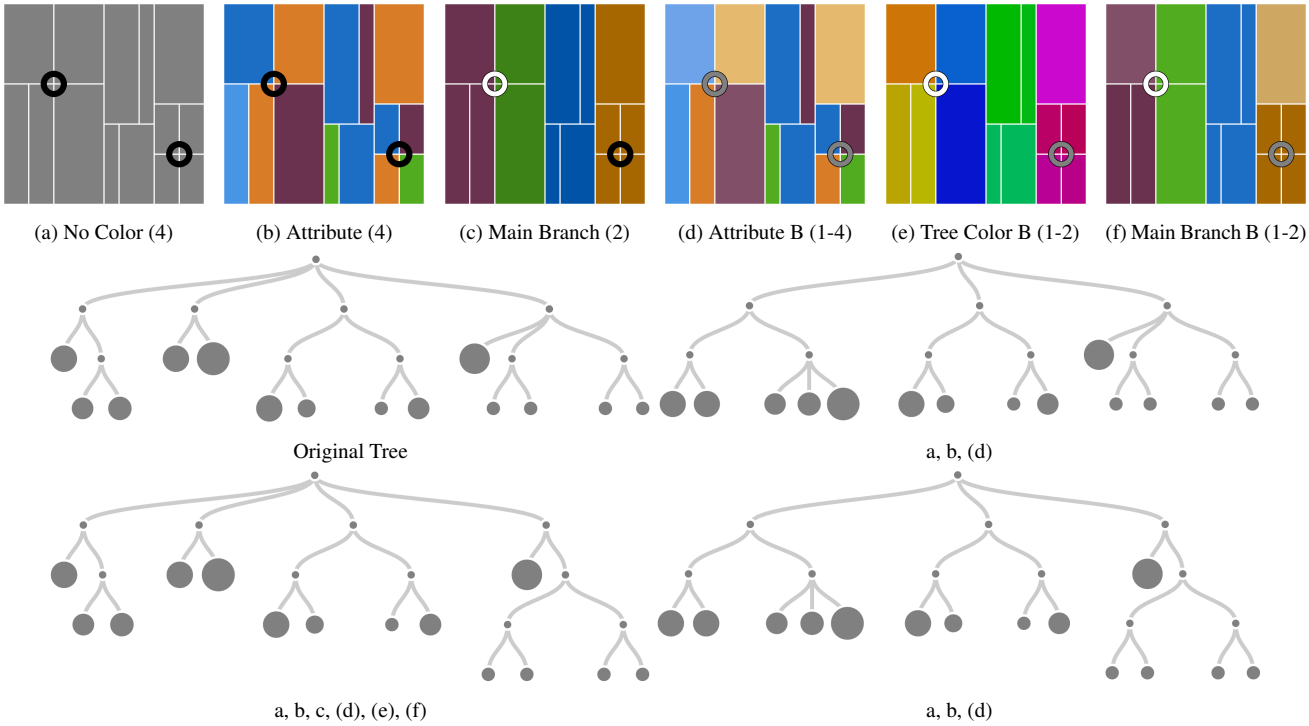



Figure 1: Treemaps colored in various color scheme with the number of reconstructed trees in parentheses. B denotes the usage of brightness to indicate node depth. Circles indicate where tree structure can be never reconstructed univocally (black), only with parameter knowledge (gray) or always (white). The first shown tree is the original tree, which will always be reconstructed. Letters beneath the other trees represent corresponding treemaps. Parentheses around letters represent where parameter knowledge is applicable.

**Abstract**

*Treemaps illustrate hierarchical data, such as file systems or budget structures. Colors are often used to encode additional information or to emphasize the tree structure. Given a treemap, one may want to retrieve the underlying data. However, treemap reconstruction is challenging, as the inner tree structure needs to be derived almost exclusively from leaf node rectangles. Furthermore, treemaps are well known to suffer from ambiguities, i.e., different input data may produce the same drawing. We present a novel reconstruction approach for slice-and-dice treemaps. Moreover, we evaluate the influence of five color schemes to resolve ambiguities. Our work can be used for the reproducibility of published data and for assessing ambiguities in slice-and-dice treemaps.*

**CCS Concepts**

• ***Human-centered computing*** → ***Treemaps;***



DOI: 10.2312/vmv.20201191

## 1. Introduction

Since the initial Slice-and-Dice (SnD) algorithm [Shn92], treemaps (TMs) have been widely used to visualize trees with node weights in space-efficient ways [BSW02; FP02; VWW99]. Today, there is an abundance of construction algorithms [Sch11] with various properties that have successfully been used on data, such as file systems [BHW00; FP02; Shn92]. Among all construction algorithms, the SnD algorithm is the most widely studied one for rectangular treemaps [Sch11]. It is also the only one that is known to preserve the hierarchical nature of the underlying tree data [TS07; BSW02].

Commonly, the visualization task ends with the final visualization of the input data. However, there are scenarios, in which the visualization is the only available source and the visualization process needs to be reversed [CLFL20]. For example, to provide visually impaired people with the reading of information contained within charts [CJP*19]. As different layouts may produce a different drawing for the same dataset [SSV18], each treemap layout requires a partially different reconstruction algorithm. As a starting point, this paper only focuses on the SnD layout, as it is the basis for all other algorithms and produces highly stable layouts.

## 2. Related Work

Both treemaps and reconstruction of data from visualizations are active fields of reasearch.

### 2.1. Treemaps

Treemap research focuses on the development of layouts for drawing treemaps [SSV18; Sch11; BB12; BSW02; BE95; TS07; BEL*11; DSF*14; WD08], rather than on reconstructing data from the treemap drawings. The proposed layouts optimize drawing quality measures [BBK*18] such as treemap readability and stability. As recent studies found that the slice-and-dice layout is very stable, both with respect to container resolution [KBW*20] and data changes [VSC*20], we focus on this layout.

Treemap drawings may be amended by additional visual clues such as color [SDW08], labels [LLWM13], cushions [VWW99] or border thickness [KHA10; BHW00]. With color being the most widely used one. It can be used to encode additional information such as stock market growth [Wat99], file types [FP02], traffic speed [SDW08] or may be random for user study purposes [KHA10]. If not used for showing additional information, it may be used for encoding tree structure. While many color schemes, also called palettes or colormaps, exist for visualization, treemap-specific colormaps are rare [ZH15]. *Tree Colors* [TJ14] is a color-scheme where nodes should have unique colors, children and parents should have similar colors and the depth of a node should be reflected in its color. More frequently used are color schemes that encode only the first level of node depth as color [TJ14; Sun15]. This is also referred to as *Main Branch Colors*. Fekete et al. [FP02] combines additional data as color with the depth of tree node as brightness. We investigate how color coding influences the unique reconstructability of tree data from the treemap drawing.

### 2.2. Data Reconstruction from Visualization

A wide variety of tools exists for data reconstruction from drawings, such as ChartSense [JKS*17], ChartDecoder [DWNZ18], WebPlotDigitizer [Roh19], DataThief [Tum16], FigureSeer [SHL*16], Revision [SKC*11] Bar Chart Extractor [AZG15], DVQA [KCPK18], Scatteract [CRMY17] or the tool by Harper et al. [HA14]. They can automatically reconstruct data from chart types such as line charts, bar charts, area charts, map charts, pie charts, radar charts, scatterplots, donuts, choropleth, marey trains, stacked bar charts or parallel coordinates. Most related to our work is PhyloParser [LYWH17]. It reconstructs phylogenetic trees from dendrograms (node-link diagrams) which show all tree elements, i.e. nodes and edges. Treemap reconstruction is more difficult as only leaf nodes are visible. Inner nodes and edges need to be derived from the visible leaf rectangles.

**Visual reconstructability – Ambiguities:** An important question in extracting data from a visualization is the precision/accuracy and uniqueness of the extracted data. Visualizations, including graph and tree drawings, may suffer from ambiguities, unfaithfulness or poor visual-reconstructability [JWC*11; NEH13; WSA*16]. The drawing may not enable the unique reconstruction of data, e.g., node and edge overplotting may lead to high ambiguities in node-link diagrams [WSA*16]. We thus investigate ambiguity of tree drawings in slice-and-dice treemaps, caused by inherent overplotting of parent nodes with child nodes.

## 3. Definitions and Terminology

**Tree:** *Tree structure:* A *directed, rooted tree*, here simply *tree*, is an acyclic graph that consists of nodes $n \in N$ connected by directed edges $e \in E$: $T = (N, E)$ with $E \subset N \times N$. It has a root node $root(T)$ and all edges are directed away from the root. The source of the directed edge $e$ is called *parent node par* and the destination is the *child node ch*. A node is called *single-child node* if it has only one child. A node is called a *leaf* $n^L$, if it has no child nodes. The set of all leaf nodes is denoted as $L(T)$. Other nodes are called *inner nodes*. The set of all inner nodes is $Inner(T)$.
*Node order*: Each node has a unique *id* defining the *node order* when traversing the tree in-order.
*Node weights:* A tree is *weighted* $T = (N, E, w)$ if all nodes have an assigned *weight* $w : N \to \mathbb{R} > 0$. The weight of a parent node is the sum of weights of its child nodes. Without loss of generality, we assume $w(root) = 1$.
*Descendant:* If a node $d$ can be reached from a node $a$, $d$ is called a *descendant* of $a$.
*Node color:* Each node can be assigned a *color* based on a *color scheme*, which is discussed at the end of this chapter.
*Node depth:* The *depth* of a node is defined as the distance between the tree's root node and the node. The root node has a depth of 0. Nodes with a depth of i are on the i-th *level* of the tree.
*Subtree:* A *subtree* of a tree $T$ is a part of $T$. It consists of a node in $T$ and all nodes and edges that are reachable from that node.

**Treemap:** A rectangular *treemap TM* is a *visual structure* displaying the input tree data $T^{IN}$ using nested, colored rectangles $R$. Each rectangle corresponds to a node in the tree $r(n) : n^{IN} \to R$. The treemap's bounding rectangle, e.g.,  corresponds to the root node.

It has the size of the treemap's target resolution $Res$. The position, width and height of a rectangle $r \in R$ are determined by the treemap visualization process $TMV$. The relative area of each rectangle corresponds with its relative node weight to the root node. Due to overplotting, the *visible rectangles* $R^{vis}$ correspond to the leaf nodes $L(T) \rightarrow R^{vis}$. The inner node rectangles are formed by a set of visible rectangles (e.g.,  ).

**Treemap Visualization Process:** The treemap visualization process TMV transforms the input tree $T^{IN} = (N^{IN}, E^{IN}, w^{IN})$ into the treemap $TM$ according to a specification $S = (LS, IS, CS)$: $TMV : T^{IN}(N^{IN}, E^{IN}, w^{IN}) \times S(LS, IS, CS) \rightarrow TM$. The layout specification $LS = (L, p)$ consists of the slice-and-dice layout $L$ [Shn92] and a parameter $p$: the starting tiling direction. The image specification $IS$ is the target resolution $Res$: the width $TMW$ and height $TMH$ of the treemap image. Color specification $CS$ is determined by the color scheme (see below).

**Treemap Reconstruction:** The visual inverse $TMV^{-1}$ of the treemap visualization process derives all possible tree structures $\{T_i^{OUT}\} = \{(N_i^{OUT}, E_i^{OUT}, w_i^{OUT})\}$ from an input treemap $TM^{IN}$. The area of a leaf rectangle determines the leaf node weight $w^{OUT}$. If applicable, it saves colors of leaf rectangles in the corresponding reconstructed leaf nodes. A treemap reconstruction algorithm is *correct* if it produces all trees that could have produced the input treemap: $\forall T_i^{OUT} \in TMV^{-1}(TM^{IN}) : TMV(T_i^{OUT}) = TM^{IN}$.

**Input, Output and Assumptions:** The algorithm reconstructs all trees $\{T_i^{OUT}\} = \{(N_i^{OUT}, E_i^{OUT}, w_i^{OUT})\}$ that could have produced an input treemap $TM^{IN}$ with the resolution $(TMW, TMH)$. The treemap is given as a set of its visible rectangles $R^{vis}$ , e.g.,  either directly in D3 [BOH11], SVG graphics, JSON data, or externally pre-processed from images. We assume that the treemap was produced with the SnD layout and a known starting direction. For an unknown starting direction, the algorithm can be run with each direction. The treemap should not have been created from trees with single-child nodes, as their rectangles are not visibly distinguishable from non-single-child node rectangles, making them neither identifiable by humans nor by algorithms.

**Color Schemes:** The way colors are assigned to nodes is defined by a *color scheme*. We use the HSV/HSB color space and assume that each node and leaf rectangle contains at most one color. Leaf rectangles containing more than one color, such as treemaps using cushions, are disregarded in this work. We focus on the following five color schemes (see Figure 1b-1f):

1. *Main Branch Colors:* All child nodes of the root of a hierarchy are assigned a unique color. Their descendants are assigned the same exact color to indicate that they belong together.
2. *Main Branch Colors with brightness:* This color scheme extends Main Branch Colors by further encoding the depth of a node. All child nodes of the root of a hierarchy are assigned a unique hue. Their descendants are assigned the same exact hue to indicate that they belong together. All nodes have their brightness set based on their depth. Nodes either get brighter or darker the deeper they are.
3. *Attribute colors:* In this color scheme, each leaf node is assigned a unique color based on a categorical attribute.
4. *Attribute colors with brightness:* This color scheme extends attribute colors by further encoding the depth of a node. Each leaf node is assigned a unique hue based on a categorical attribute. Similarly to Main Branch Colors with brightness, nodes either become brighter or darker the deeper they are.
5. *Tree Colors:* Tree Colors [TJ14] is a color scheme specifically designed for hierarchical data. It also sets nodes' brightness values based on depth; nodes either become darker or brighter the deeper they are. Additionally, Tree Colors has two more goals. First, each node should have a unique color. Secondly, the relation between parent nodes and child nodes needs to be obvious through the colors. These properties are achieved by assigning equally big hue portions from 0-360 to sibling nodes. Child nodes recursively split their parents' hue ranges, which causes them to have similar colors. Tree Colors require different parameter settings such as the hue *fraction* or whether hue ranges should be *permuted* or *reversed*. Changing these parameters can make it easier to distinguish nodes that don't belong together. Due to the large parameter space, treemaps with identical structures can look very different.

## 4. Reconstruction

In Section 4.1 we present a general but straightforward approach to reconstruct tree structures and weights from SnD treemaps, irrespective of colors. In Section 4.2 and 4.3, we explain how we optimized the memory consumption and processing time to increase the performance. We then prove the algorithm's correctness in Section 4.4. Finally, in Section 4.5 we explain how we can reduce the resulting number of trees using color schemes. Note that colors do not change rectangle positions and therefore never increase the ambiguity of a treemap.

### 4.1. General Reconstruction Approach

**Algorithm 1** Initial reconstruction algorithm that converts a treemap into a set of trees.

**Input:** $R = R^{vis} = \{r_1, r_2, ...r_n\}$ ▷ Treemap rectangles
**Output:** $TT = TT^{OUT} = \{T_1, T_2, ...T_m\}$ ▷ A set of output trees
**function** GENERATETREES($R$)
  **if** $|R| = 1$ **then return** $\{R\}$ ▷ Return rect as leaf node.
  $SD \leftarrow$ GENERATEALLSUBDIVISIONS($R$)
  $TT \leftarrow \emptyset$
  **for all** $sd \in SD$ **do**
    $RR_{sd} \leftarrow$ SPLITRECTANGLES($R$, $sd$)
    $TT_{sd} \leftarrow \{\emptyset, \emptyset\}$ ▷ Single tree without children
    **for all** $R_{sd} \in RR_{sd}$ **do**
      $TT_{sd} \leftarrow TT_{sd} \times$ GENERATETREES($R_{sd}$)
    **end for**
    $TT \leftarrow TT + TT_{sd}$
  **end for**
  **return** $TT$
**end function**

The general algorithm GENERATETREES reconstructs the tree

structure in a top-down manner from the root by adding child nodes and their descendants recursively down to the leaves (see Algo. 1). We find possible input trees by using the SnD algorithm's specific tiling strategy on the visible rectangles $R = R^{vis}$.

If $R$ contains a single rectangle, the algorithm returns a set containing a single tree with only this rectangle as root node (line 2). Otherwise, we need to create all subdivisions $SD$ of the current set of rectangles (line 3). Subdivisions can be found with the help of *separating lines* $SL$. These lines fully tile a rectangle along a subdivision direction (e.g., ). An $SL$ is either a line directly created by the SnD construction algorithm, or a combination of two (or more) lines created in a later construction step  or . A rectangle can have several possible subdivisions . A particular subdivision $sd \in SD$ is determined by a subset of separating lines $SL_{sd} \subset SL$ and splits the corresponding rectangles $R^{vis}$ at each separating line in $SL_{sd}$ into $RR^{vis}_{sd} = \{R^{vis}_1 ... R^{vis}_i\}$ (line 6). Subdivisions are *(in)valid*, if they can (cannot) be formed by the SnD layout, e.g., .

If there was no subdivision possible, the algorithm will return an empty set (due to $TT \leftarrow \emptyset$ in line 4), signalling that there was no possible input tree that could produce the set of rectangles $R$ with the SnD algorithm. Otherwise, we need to build subtrees for each subdivision $sd \in SD$ by splitting the rectangles according to the split lines (line 6) into a set $RR^{vis}_{sd}$ containing sets of rectangles.

Once all subtrees have been created, we need to form the crossproduct between them for each $R^{vis}_{sd} \in RR^{vis}_{sd}$ (line 9) as the construction of the subtrees is independent of each other. Note that if any one set of rectangles $R_{sd}$ did not result in a reconstructed tree, the crossproduct will automatically result in $TT_{sd} = \emptyset$, discarding all trees built for all other $R'^{vis}_{sd} \in RR^{vis}_{sd}$. The trees for the current subdivision will be appended to the list of trees (line 11) and returned after all subdivisions were processed.

After the structure of the input tree has been recovered, the weight of a child node is calculated as $w = {Area(R^{ch})}/{Area(R^{vis})}$. Note that child nodes may be shared between several subdivisions if they correspond to the same parent rectangle, e.g. .

### 4.2. Memory Consumption Optimization

**Maximum Ambiguity – Memory Consumption:** The general algorithm generates each reconstructed tree as an individual copy. To illustrate the implications, we will first introduce the *maximum possible ambiguity* and show the found maximum ambiguities in our evaluation. We will then introduce a solution to reduce the required memory. As mentioned but not quantified in [BHW00; Eic97], the theoretical maximum structural ambiguity is reached for a treemap shaped like a regular grid. In Equation 1, we present a method to calculate the ambiguity $SA\left(R^{grid}_{j,k}\right)$ of a grid shaped treemap with $j \times k = \left|R^{grid}_{j,k}\right| = \left|R^{vis}\right|$ leaf nodes. Note that we only need to calculate the ambiguity for a horizontal starting direction, as the ambiguity for the vertical starting direction can be calculated by swapping $j$ and $k$. We split our calculation into the first child node and the remainder of the root node and iterate over all possible grid sizes, i.e. $1 \times k$ to $(j-1) \times k$. Since we need to switch from the horizontal to vertical split direction, the ambiguity for the first child node can be calculated as $SA\left(R^{grid}_{k,i}\right)$ with $k = [1..j-1]$. For the remainder, we can either continue by splitting the visible rectangles into further child nodes, or by adding a single node containing all visible rectangles. The ambiguity for the first case is $SA\left(R^{grid}_{j-i,k}\right)$ since the split direction remains unchanged. For the second case, we again calculate the ambiguity by switching the horizontal and vertical number of rectangles, i.e. $SA\left(R^{grid}_{k,j-i}\right)$. If there is, however, only a single visible rectangle left, i.e. $j - i = k = 1$, adding an extra child node for a single rectangle is not allowed. We therefore introduce $SA_c$ which is the same as $SA$ except for both $j = 1$ and $k = 1$. Note that all other $SA\left(R^{grid}_{1,k}\right)$ are 0 due to the empty sum as they do not represent a valid treemap for the chosen horizontal starting direction.

$$
\begin{aligned}
SA\left(R^{grid}_{1,1}\right) = 1 \quad SA_c\left(R^{grid}_{1,1}\right) = 0 \\
SA\left(R^{grid}_{j,k}\right) = SA_c\left(R^{grid}_{j,k}\right) = \\
\sum_{i=1}^{j-1} SA\left(R^{grid}_{k,i}\right) \cdot \left(SA\left(R^{grid}_{j-i,k}\right) + SA_c\left(R^{grid}_{k,j-i}\right)\right)
\end{aligned}
\tag{1}
$$

| Nr. leaves | 25 | 50 | 100 | 200 | 400 | 800 |
|---|---|---|---|---|---|---|
| Th. max. str. amb. | 537 | $3 \cdot 10^8$ | $3 \cdot 10^{16}$ | $6 \cdot 10^{37}$ | $5 \cdot 10^{78}$ | $7 \cdot 10^{155}$ |
| Max eval amb. | 4 | 8 | 32 | 512 | 3,072 | 16,515,072 |

Table 1: Theoretic maximum ambiguity (rounded values), as well as the maximum ambiguity found in our evaluation for a weight distribution of 1-20. See Figure 6 for more detailed distributions.

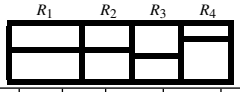
Table 1 shows that theoretical maximum ambiguity and the maximum ambiguity from our evaluation. Both increase super-exponentially with the number of leaves, i.e., $|R^{vis}|$ in a treemap. While the chance of a perfect grid and theoretical maximum ambiguity is low, over 16 million trees for single treemaps can still be challenging especially when considering large datasets.

**Optimization:** To reduce the memory consumption, we implement a tree structure where every node contains a set of possible subtrees instead of child nodes. Only one of those trees is needed for all reconstructed trees for a single treemap, eliminating duplicates that occur when creating individual trees. Furthermore, subtrees for identical $R^{vis}$ and same tiling direction that are needed more than once only need to be reconstructed and stored once when using references (as explained in chapter 4.3), leading to the elimination of duplicated data in memory.

### 4.3. Runtime Optimization

The general reconstruction algorithm checks power sets of potential trees for correctness in nearly every recursion. To emphasize this problem, we introduce the following example:

Figure 2 has $|P(sd)| = 2^3 - 1 = 7$ possible subdivisions for $R_1, R_2, R_3, R_4$. We assume that $R_2R_3$ and $R_3R_4$ are combinations that do not result in reconstructed trees, while $R_1R_2$ results in a reconstructed tree. For all 7 subdivisions : $\{R_1, R_2, R_3, R_4\}$,




| $R_{comb}$ | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_{1,2}$ | $R_{2,3}$ | $R_{3,4}$ | $R_{1,2,3}$ | $R_{2,3,4}$ |
|---|---|---|---|---|---|---|---|---|---|
| general: | 4 | 2 | 2 | 4 | 2 | 1 | 2 | 1 | 1 |
| improved: | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

Figure 2: Example treemap and subtree calculation table.

$\{\{R_1,R_2\},R_3,R_4\}$, $\{R_1,\{R_2,R_3\},R_4\}$, $\{R_1,R_2,\{R_3,R_4\}\}$, $\{\{R_1,R_2\},\{R_3,R_4\}\}$, $\{\{R_1,R_2,R_3\},R_4\}$, $\{R_1,\{R_2,R_3,R_4\}\}$ the general algorithm creates the subtrees listed in Figure 2. Each potential subtree has to be processed separately from this point on, while only the subdivisions $\{R_1,R_2,R_3,R_4\}$ and $\{\{R_1,R_2\},R_3,R_4\}$ lead to reconstructed trees. The problem of duplicate calculations is shown in Table 1. Furthermore, each node's child nodes result in a power set as well and ambiguous structures lead to additional sets that need to be checked, as the tiling direction in further sets of $R$ may change and the number of different sets of $R$ in subtrees increases. As a base for a more efficient reconstruction algorithm, we first define the primetree $T'$ of a treemap $TM$ as following:

**Theorem 1 (primetree)** If a treemap $TM$ was created using the SnD algorithm, we can construct a tree $T'$ (primetree) that creates the treemap $TM$ while always using the maximum subdivision, where the maximum subdivision is the one that splits the set of rectangles into the maximum number of children.

*Proof* Given a set of rectangles $R$ and the maximum subdivision $sd$, there are two possible cases for the SnD construction algorithm:

1. A child contains only a single rectangle and is therefore a leaf node and is therefore a valid tree.
2. Further subdivision of the children is possible according to the SnD layout rules.

In case of 2., we can again split the child using the maximum subdivision until all children are leaf nodes. □

Using the prime tree definition, we can now use the following recursive approach to improve the general reconstruction algorithm:

1. We create the prime tree $T'$ for all $R^{vis}$ contained in a bounding rectangle (starting with all $R^{vis}$). This is possible for all SnD treemaps as shown in Theorem 1.
2. We traverse $T'$ and enumerate all subdivisions that lead to reconstructed trees for each node's child nodes. We then repeat both steps for these new subdivisions.

During the prime tree creation, we save all $SL$ in their respective nodes in the prime tree and can then find subdivisions that lead to reconstructed trees by matching the $SL$ of sibling nodes during tree traversal. No further $SL$ detection is needed and all found subdivisions will lead to prime trees for their subsets of $R^{vis}$, as the subsets were created with the SnD construction algorithm (Theorem 1). Therefore, when traversing $T'(\{R_1,R_2,R_3,R_4\})$ from the previous example, we will only find $\{\{R_1,R_2\},R_3,R_4\}$ and no further tree creation for other subdivisions is needed. During the creation of the prime tree, we already created the subtrees for $R_1,R_2,R_3$ and $R_4$ and do not need to calculate them again. Furthermore, we use the enumeration method proposed in Equation 1 to reduce the calculation cost. As this is a sequential approach, we can reuse once created subtrees ($\{R_1,R_2\}$) in later tree reconstructions. As seen in Table 2, we greatly decreased the number of needed tree reconstructions. The number of subtree creations needed by the general algorithm increases exponentially with the number of crossproduct combinations. The number of subtree creations needed by the improved algorithm is either 1 or 0 for every possible subtree. Therefore, the runtime decreases from exponential to linear. No tree data reconstruction took longer than three seconds.

### 4.4. Proof of Correctness and Completeness

Due to the improvements, we can now prove that our algorithm finds only but all correct trees. Note that during the proof, without loss of generality, we assume that the input tree weight of each leaf node is the same as the number of pixels of the corresponding leaf rectangle in the treemap, i.e. any potential rounding and scaling of weights is assumed to happen prior to the construction. There are two possibilities for the origin of a separating line $sl$:

1. $sl$ is a separating line, resulting from the SnD construction algorithm.
2. $sl$ is the result of two or more separating lines from different nodes, creating a single separating line for their common immediate parent node.

The algorithm is *correct* iff the following theorem holds:

**Theorem 2 (correctness)** For all trees $T$ and treemaps $TM$, the tree $T$ is part of the reconstruction set $TM$ ($T \in TMV^{-1}(TM)$) iff the construction algorithm creates the treemap $TM = TMV(T)$, i.e. $\forall(T,TM) : TM = TMV(T) \iff T \in TMV^{-1}(TM)$

Theorem 2 is equivalent to the following two corollaries:

**Corollary 2.1 (necessary condition of Theorem 2)** If a treemap $TM$ was created with the SnD algorithm and an input tree $T$, $T$ is in the set of trees found by the reconstruction algorithm. $\forall(T,TM) : TM = TMV(T) \Rightarrow T \in TMV^{-1}(TM)$

*Proof* Assume, $\exists(T,TM) : TM = TMV(T) \wedge T \notin TMV^{-1}(TM)$. It means there exists a rectangle and a set of split lines that were created during construction but not considered during reconstruction. The reconstruction algorithm always tests all subdivisions, including the one that was formed by the SnD algorithm applied to the input tree $T$. It will therefore always reconstruct the tree $T$ which contradicts the assumption, thus the Corollary 2.1 holds. □

**Corollary 2.2 (sufficient condition of Theorem 2)** If a tree $T$ was found with the reconstruction algorithm applied to the treemap $TM$, the treemap can be re-created with the SnD algorithm: $\forall(T,TM) : T \in TMV^{-1}(TM) \Rightarrow TM = TMV(T)$

*Proof* Assume, $\exists(T,TM) : T \in TMV^{-1}(TM) \wedge TM \neq TMV(T)$. It means that the reconstruction algorithms finds a tree $T$ that does not re-create the treemap $TM$ with the SnD algorithm. However, the SnD algorithm always creates the same lines that were used during reconstruction and therefore the same treemap. Since this contradicts the assumption, the Corollary 2.2 and, therefore, the Theorem 2 holds. □

### 4.5. Reconstruction with Colors

After the reconstruction of all possible trees $T^{OUT}$ for a treemap $TM^{IN}$, we can reduce the number of valid trees by rejecting those that do not match the input color scheme. To reject a tree, we verify that the color properties of each reconstructed tree $T_i^{OUT}$ match the input color scheme. If at any reconstruction step, any of the conditions are violated, we conclude that $T_i^{OUT}$ does not match the color scheme, thus it is not the original tree $T^{IN}$ used to create $TM^{IN}$. The conditions depending on the color scheme are:

1. Main Branch Colors:
   a. All descending leaf nodes of the same child of the root must have the same exact color.
   b. Descending leaf nodes of different children of the root must have different colors.
2. Main Branch Colors with brightness:
   a. All descending leaf nodes of the same child of the root must have a similar hue. To account for rounding errors, we accept colors whose hue values differ by no more than 2.
   b. Descending leaf nodes of different children of the root must have hues that differ enough. We require those hues to differ by at least 3 to account for rounding errors.
   c. Brightness: For each child node of the root, we verify both of the following conditions:
      i. All descending leaf nodes that are on the same level need to have the same brightness.
      ii. The brightness values of descending leaf nodes on different levels must consistently decrease or consistently increase, depending on how the brightness was calculated.
3. Attribute colors:
   Currently, there is no suitable approach for attribute colors, as they do not encode any information about the hierarchy's structure. Neither the hues nor the brightness or saturation of the colors are related to the structure, which means that we cannot reject any of the reconstructed trees.
4. Attribute colors with brightness:
   While pure attribute colors do not reduce treemap ambiguity, the addition of brightness based on depth may aid the reconstruction. We verify the following conditions:
   a. All leaf nodes that are on the same level need to have the same brightness.
   b. The brightness values of leaf nodes on different levels must consistently decrease or consistently increase, depending on how the brightness was calculated.
5. Tree Colors:
   Tree Colors scheme has a large parameter space, thus we need to define general conditions that are parameter independent. We verify the following properties:
   a. All leaf nodes that are on the same level need to have the same brightness.
   b. The brightness values of leaf nodes on different levels must consistently decrease or consistently increase.
   c. We recursively calculate the average hues of inner nodes by averaging the hue values of their child nodes. Because Tree Colors assign the root node a hue range of 0-360, we can expect the original tree's root to have an average hue of around 180. Due to rounding errors, we accept trees if their root's average hue is between 175 and 185. The parameters for hue *permutation* or *reversal* do not impact this approach as they only change the order of hues.
   d. Optional: If we know that the treemap was colorized with the parameters *permutation* and *reversal* set to *false*, we can check if all leaf nodes' hues are in order. We traverse the tree in-order and verify that the hue of each leaf node is greater or equal to the last found leaf's hue $-1$, accounting for possible rounding errors.

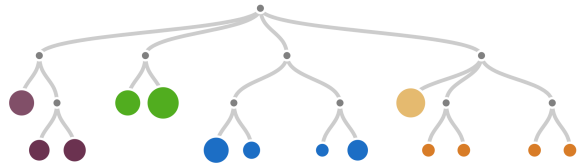
As each tree has to be checked individually, the runtime scales directly with the number of reconstructed trees. For our largest number of reconstructions with $16,515,072$ trees, each different color scheme procedure took about one hour. Each individual tree in this example took less than 0.3ms.

**Algorithm 2** The algorithm to accept or reject reconstructed trees for Main Branch Colors with Brightness.

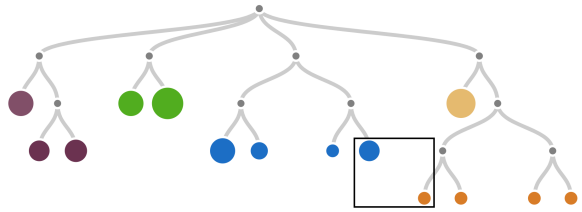
```
      Input: T                                  ▷ Reconstructed Tree
      Output: valid  ▷ Whether the tree fits the scheme
 1: function CHECKMAINBRANCHBRIGHTNESS(T)
 2:     valid ← true
 3:     for all ch ∈ children(root(T)) do
 4:         if valid = false then break
 5:         valid ← SIMILARLEAFHUES(ch)
 6:     end for
 7:     if valid = true then
 8:     valid ← DIFFERENTHUESINMAINBRANCHES(T)
 9:     for all ch ∈ children(root(T)) do
10:         if valid = false then break
11:         brightnessValues ← new double [MAXDEPTH(T) +1]
12:         valid ← SAMEBRIGHTNESS(ch, brightnessValues)
13:         if valid = true then
14:         valid ← CHANGINGBRIGHTNESS(brightnessValues)
15:     end for
16:     return valid
17: end function
```

Algo. 2 shows the steps discussed earlier for Main Branch Colors with Brightness based on depth. First, we verify the hue-based properties using SIMILARLEAFHUES (line 5) and DIFFERENTHUESINMAINBRANCHES (line 8). Afterwards, SAMEBRIGHTNESS checks for each child of the root whether descending leaf nodes on the same level have the same brightness (line 12). Finally, CHANGINGBRIGHTNESS makes sure that brightness values on different levels consistently decrease or consistently increase (line 14). Note how the brightness-based checks are done separately for each main branch (line 9); i.e. we do not compare brightness values of different main branches, as each main branch may have started with a different brightness.
Also note how if at any point any of the conditions are violated, *valid* is set to *false* and we skip all further checks to immediately reject the tree.

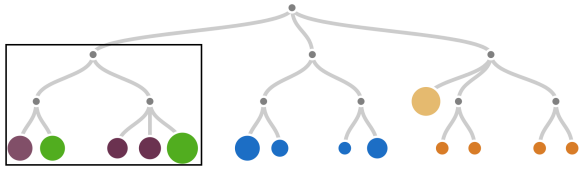
Figure 3 shows the four trees that can be reconstructed from the treemap in Figure 1f. They are structurally identical to the trees from Figure 1, but nodes are colored like their corresponding rectangles. Trees 3c and 3d can be rejected because their left main branches contain both purple and green (see highlighted subtrees). Trees 3a and 3b both satisfy all conditions and cannot be rejected.



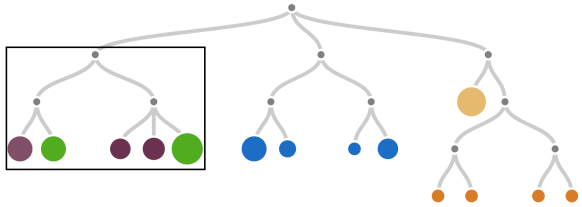
(a) Tree is accepted.



(b) Tree can be rejected if main branches start with same brightness.



(c) Tree is rejected.



(d) Tree is rejected.

Figure 3: Colored Trees

However, if we also knew that all main branches started with the same brightness, we could apply the brightness-based checks to the entire tree, rather than for each main branch separately. In that case we could also reject tree 3b, because the highlighted nodes are on different levels, yet contain the same brightness.

## 5. Evaluation

We used our reconstruction and color refinement algorithms to quantify the ambiguity and how well colorization can decrease ambiguity in SnD treemaps. For our evaluation, we randomly generated 12,000 trees and constructed 60,000 treemaps with a set of predefined parameters for the colorization and tree generation.

### 5.1. Data Generation and Treemap Construction

We generated the tree data for our evaluation with two leaf node weight ranges {(1-20),(5-100)} and six leaf node counts {25, 50, 100, 200, 400, 800}. Starting from the root, every inner node has, randomly determined, between two and five child nodes, eliminating the possibility of single-child nodes. The depth varies depending on the randomized generation, but is bound due to the maximum number of leaf nodes. For each setting we generated 1,000 trees resulting in 2×6×1,000 = 12,000 trees. These settings were inspired by literature [KBW*20; Shn92; SW01; BSW02]. We then randomly applied colors following our color schemes on each tree for each color scheme and visualized the trees as treemaps with an 800×800px resolution. This resulted in 60,000 treemaps. The generated treemaps were reconstructed with our proposed algorithm to assess the impact of ambiguity on SnD treemaps. We then applied our color checks on each reconstruction to evaluate how well each color scheme can diminish ambiguity.

### 5.2. Color settings

We use the HSV/HSB color space for all calculations. Unless noted otherwise, we set the *saturation* and *brightness/value* to 1.

1. Main Branch Colors:
   - Hue: We use the hues {0, 30, 60, 120, 180} for children of the root.
2. Main Branch Colors with brightness:
   - Hue: We use the hues {0, 30, 60, 120, 180} for children of the root.
   - Value/Brightness: We calculate the brightness as $0.9^{depth-1}$.
3. Attribute colors:
   - Hue: We use the hues {0, 30, 60, 120, 180} for attributes.
4. Attribute colors with brightness:
   - Hue: We use the hues {0, 30, 60, 120, 180} for attributes.
   - Value/Brightness: We calculate the brightness as $0.9^{depth-1}$.
5. Tree Colors:
   - Hue: The root begins with a hue range of 0-360. Children recursively split their parents' hue ranges in equal sub-ranges, hue values are chosen as the middle of a hue range.
   - Saturation: We calculate the saturation as $1-0.2^{depth}$.
   - Value/Brightness: We calculate the brightness as $0.9^{depth-1}$.
   - Parameters: We set the fraction to 0.5, enabled hue permutation and disabled hue reversal.

   As Tennekes [TJ14] noted, the settings for the color scheme depend on multiple factors such as the size of the hierarchy, how deep it is and on which level the attention is focused. We chose these settings because they make sense for evaluation with varying leaf node counts.

### 5.3. Evaluation Results

We show the results in a separate visualization for each weight range. Figure 4 shows the percentage of SnD treemaps that are ambiguous without taking color schemes into consideration. While the larger leaf node weight range has a diminishing effect on the occurrence of ambiguity, the effect weakens for larger leaf node counts. For both weight ranges, treemaps with 100 or more leaf nodes have more than 40% chance of being ambiguous. Treemaps with 400 or more leaf nodes, the chance for a non-ambiguous treemap is under 2%.

Figure 5 shows the percentage of ambiguous treemaps, where ambiguity could be perfectly resolved by a color scheme. Note that the attribute color scheme without brightness never reduced ambiguity and is therefore not shown. While the main branch color scheme is not sufficient to reliably resolve ambiguities, the main branch brightness, attribute brightness and tree colors color

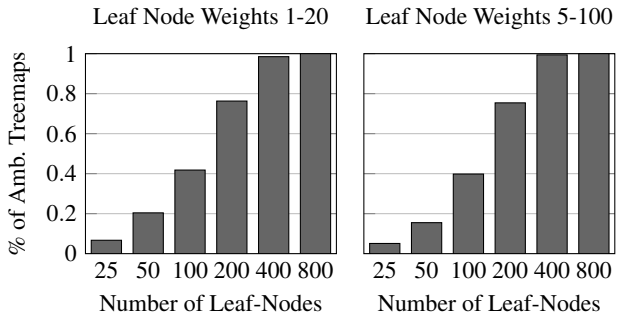



Figure 4: Percentage of ambiguous treemaps in our data set.

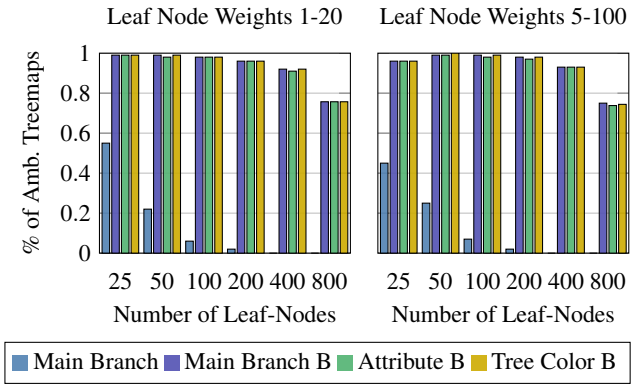



Figure 5: Percentage of ambiguous treemaps that could be resolved perfectly with the help of color schemes. B denotes the usage of brightness to indicate node depth.

schemes manage to perfectly resolve the ambiguity in over 90% of cases for treemaps with 400 or fewer leaf nodes.

Figure 6 shows the distribution of ambiguity for treemaps without colors and the distributions after applying the ambiguity reduction process for each color scheme. This boxplot emphasizes that the amount of reconstructed trees grows super-exponentially with increasing leaf nodes even for non-grid like treemaps. The largest amount of 16,515,072 reconstructed trees for a single treemap was found for a tree with 800 leaf nodes and a weight range of 1-20. Furthermore, the main branch brightness, attribute brightness and tree color color schemes can reduce even large amounts of ambiguity in treemaps. They could even reduce the largest ambiguity to only the original input tree. Color schemes that do not incorporate brightness to encode depth did not significantly reduce ambiguity.

## 6. Conclusion and Future Work

We reconstructed the potentially ambiguous tree data from SnD treemaps and evaluated how five color schemes can help to reduce occurring ambiguity. In our evaluation, we found that ambiguity in SnD treemaps grows super-exponentially with the number of leaf nodes. 40% of treemaps produced from trees with 100 leaf nodes

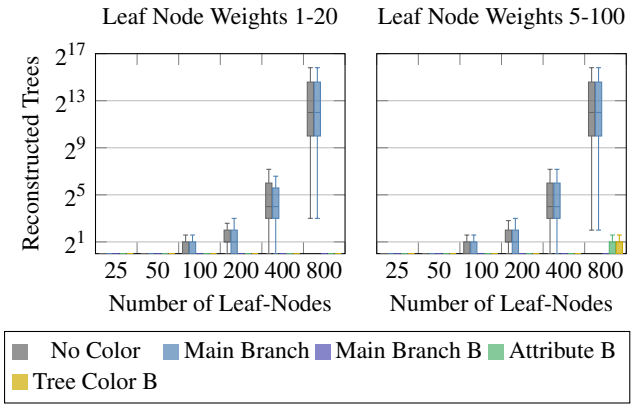



Figure 6: Distributions of ambiguity in each evaluation set. B denotes the usage of brightness to indicate node depth.

are ambiguous. At 800 leaf nodes, treemap reconstruction can result in over ten million reconstructed trees and not a single treemap without ambiguity was found. When incorporating color schemes into our reconstruction process, we found that the usage of brightness to encode node depth can vastly reduce ambiguity, independent of color hues. For tree data with additional attributes, we suggest using the *attribute with brightness* color scheme. When drawing treemaps for tree data without attribute information, we suggest using *main branch colors with brightness* or *tree colors*. However, tree colors are difficult to use as good parameters need to be chosen for each specific dataset.

In the future, more reconstruction algorithms need to be researched to compare the reconstructability of different layouts. Furthermore, there are additional color schemes that might be useful to explore. The automatic reconstruction of colormaps [PMH18] for treemaps can be a great addition for future work. In this context, a parameter evaluation for tree colors can be interesting to find the optimal settings for treemap reconstruction. Finally, the ability of humans to read ambiguous treemaps and understand colors in comparison to automatic reconstructions is another interesting direction of future research.

## References


[AZG15] AL-ZAIDY, RABAH A. and GILES, C. LEE. "Automatic Extraction of Data from Bar Charts". *K-CAP*. 2015 2.

[BB12] BAUDEL, THOMAS and BROEKSEMA, BERTJAN. "Capturing the design space of sequential space-filling layouts". *IEEE TVCG* 18.12 (2012) 2.

[BBK*18] BEHRISCH, MICHAEL, BLUMENSCHEIN, MICHAEL, KIM, NAM WOOK, et al. "Quality metrics for information visualization". *Computer Graphics Forum* 37.3 (2018) 2.

[BE95] BAKER, MARLA J. and EICK, STEPHEN G. "Space-filling Software Visualization". *Journal of Visual Languages and Computing* 6.2 (1995). DOI: 10.1006/jvlc.1995.1007 2.

[BEL*11] BUCHIN, KEVIN, EPPSTEIN, DAVID, LÖFFLER, MAARTEN, et al. "Adjacency-Preserving Spatial Treemaps". *International Symposium on Algorithms and Data Structures*. Lecture Notes in Computer Science. Springer, 2011. DOI: 10.1007/978-3-642-22300-6{_}14 2.

[BHW00] BRULS, MARK, HUIZING, KEES, and van WIJK, JARKE J. "Squarified Treemaps". *Data Visualization*. Springer, 2000. ISBN: 978-3-7091-6783-0 2, 4.

[BOH11] BOSTOCK, MICHAEL, OGIEVETSKY, VADIM, and HEER, JEFFREY. "$D^3$ data-driven documents". *IEEE TVCG* 17.12 (2011) 3.

[BSW02] BEDERSON, BENJAMIN B., SHNEIDERMAN, BEN, and WATTENBERG, MARTIN. "Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies". *ACM TOG* 21.4 (2002). DOI: 10.1145/571647.571649 2, 7.

[CJP*19] CHOI, JINHO, JUNG, SANGHUN, PARK, DEOK GUN, et al. "Visualizing for the Non-Visual: Enabling the Visually Impaired to Use Visualization". *Computer Graphics Forum*. Vol. 38. 3. 2019 2.

[CLFL20] CHAI, CHENGLIANG, LI, GUOLIANG, FAN, JU, and LUO, YUYU. "CrowdChart: Crowdsourced Data Extraction from Visualization Charts". *IEEE TKDE* (2020) 2.

[CRMY17] CLICHE, MATHIEU, ROSENBERG, DAVID, MADEKA, DHRUV, and YEE, CONNIE. "Scatteract: Automated Extraction of Data from Scatter Plots". *ECML/PKDD*. 2017 2.

[DSF*14] DUARTE, FELIPE SLG, SIKANSI, FABIO, FATORE, FRANCISCO M, et al. "Nmap: A novel neighborhood preservation space-filling algorithm". *IEEE TVCG* 20.12 (2014) 2.

[DWNZ18] DAI, WENJING, WANG, MENG, NIU, ZHIBIN, and ZHANG, JIAWAN. "Chart decoder: Generating textual and numeric information from chart images automatically". *Journal of Visual Languages & Computing* 48 (2018) 2.

[Eic97] EICK, SG. *Visualization and interaction techniques*. 1997 4.

[FP02] FEKETE, J. D. and PLAISANT, C. "Interactive information visualization of a million items". *IEEE InfoVis*. 2002. DOI: 10.1109/INFVIS.2002.1173156 2.

[HA14] HARPER, JONATHAN and AGRAWALA, MANEESH. "Deconstructing and restyling D3 visualizations". *ACM symposium on User interface software and technology*. ACM. 2014 2.

[JKS*17] JUNG, DAEKYOUNG, KIM, WONJAE, SONG, HYUNJOO, et al. "Chartsense: Interactive data extraction from chart images". *CHI Conference on Human Factors in Computing Systems*. ACM. 2017 2.

[JWC*11] JÄNICKE, HEIKE, WEIDNER, THOMAS, CHUNG, DAVID, et al. "Visual Reconstructability As a Quality Metric for Flow Visualization". *EuroVis*. The Eurographs Association & John Wiley & Sons, Ltd., 2011. DOI: 10.1111/j.1467-8659.2011.01927.x 2.

[KBW*20] KNAUTHE, VOLKER, BALLWEG, KATHRIN, WUNDERLICH, MARCEL, et al. "Influence of Container Resolutions on the Layout Stability of Squarified and Slice-And-Dice Treemaps". *EuroVis - Short Papers*. The Eurographics Association, 2020. DOI: 10.2312/evs.20201055 2, 7.

[KCPK18] KAFLE, KUSHAL, COHEN, SCOTT, PRICE, BRIAN, and KANAN, CHRISTOPHER. "DVQA: Understanding data visualizations via question answering". *IEEE CVPR*. 2018 2.

[KHA10] KONG, NICHOLAS, HEER, JEFFREY, and AGRAWALA, MANEESH. "Perceptual guidelines for creating rectangular treemaps". *IEEE TVCG* 16.6 (2010) 2.

[LLWM13] LU, H., LIU, S.X., WATTENBERG, M.M., and MA, X.J. *Constructing a labeled treemap with balanced layout*. 2013. URL: https://www.google.com.pg/patents/US8443281 2.

[LYWH17] LEE, PO-SHEN, YANG, SEAN T., WEST, JEVIN D., and HOWE, BILL. "PhyloParser: A Hybrid Algorithm for Extracting Phylogenies from Dendrograms". *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)* 01 (2017) 2.

[NEH13] NGUYEN, Q., EADES, P., and HONG, S. H. "On the faithfulness of graph visualizations". *IEEE PacificVis*. IEEE. 2013 2.

[PMH18] POCO, JORGE, MAYHUA, ANGELA, and HEER, JEFFREY. "Extracting and retargeting color mappings from bitmap images of visualizations". *IEEE TVCG* 24.1 (2018) 8.

[Roh19] ROHATGI, ANKIT. *WebPlotDigitizer*. https://automeris.io/WebPlotDigitizer/. accessed on April 2019. 2019 2.

[Sch11] SCHULZ, H. "Treevis.net: A Tree Visualization Reference". *IEEE CG&A* 31 (Nov. 2011). DOI: 10.1109/MCG.2011.103 2.

[SDW08] SLINGSBY, AIDAN, DYKES, JASON, and WOOD, JO. "Using treemaps for variable selection in spatio-temporal visualisation". *Information Visualization* 7.3-4 (2008) 2.

[SHL*16] SIEGEL, NOAH, HORVITZ, ZACHARY, LEVIN, ROIE, et al. "FigureSeer: Parsing Result-Figures in Research Papers". *ECCV*. 2016 2.

[Shn92] SHNEIDERMAN, BEN. "Tree Visualization with Tree-maps: 2-d Space-filling Approach". *ACM Trans. Graph.* 11.1 (1992). DOI: 10.1145/102377.115768 2, 3, 7.

[SKC*11] SAVVA, MANOLIS, KONG, NICHOLAS, CHHAJTA, ARTI, et al. "Revision: Automated classification, analysis and redesign of chart images". *ACM UIST*. ACM. 2011 2.

[SSV18] SONDAG, MAX, SPECKMANN, BETTINA, and VERBEEK, KEVIN. "Stable Treemaps via Local Moves". *IEEE TVCG* 24.1 (2018). DOI: 10.1109/TVCG.2017.2745140 2.

[Sun15] SUN, SHIPENG. "A perception-based color recommendation algorithm for hierarchical regions". *Cartography and Geographic Information Science* 42.3 (2015) 2.

[SW01] SHNEIDERMAN, BEN and WATTENBERG, MARTIN. "Ordered treemap layouts". *Information Visualization, 2001. INFOVIS 2001. IEEE Symposium on*. IEEE. 2001 7.

[TJ14] TENNEKES, MARTIJN and de JONGE, EDWIN. "Tree colors: color schemes for tree-structured data". *IEEE TVCG* 20.12 (2014) 2, 3, 7.

[TS07] TU, YING and SHEN, HAN-WEI. "Visualizing changes of hierarchical data using treemaps". *IEEE TVCG* 13.6 (2007) 2.

[Tum16] TUMMERS, B. *Data Thief*. http://datathief.org. 2016 2.

[VSC*20] VERNIER, EDUARDO, SONDAG, MAX, COMBA, JOÃO, et al. "Quantitative Comparison of Time-Dependent Treemaps". *CGF* (2020). DOI: 10.1111/cgf.13989 2.

[VWW99] VAN WIJK, JARKE J. and van de WETERING, HUUB. "Cushion Treemaps: Visualization of Hierarchical Information". *IEEE Symposium on Information Visualization*. INFOVIS '99. IEEE Computer Society, 1999. ISBN: 0-7695-0431-0. URL: http://dl.acm.org/citation.cfm?id=857189.857663 2.

[Wat99] WATTENBERG, MARTIN. "Visualizing the stock market". *CHI extended abstracts on Human factors in computing systems*. ACM. 1999 2.

[WD08] WOOD, J. and DYKES, J. "Spatially Ordered Treemaps". *IEEE TVCG* 14 (2008). DOI: 10.1109/TVCG.2008.165 2.

[WSA*16] WANG, Y., SHEN, Q., ARCHAMBAULT, D., et al. "AmbiguityVis: Visualization of Ambiguity in Graph Layouts". *IEEE TVCG* 22.1 (2016). DOI: 10.1109/TVCG.2015.2467691 2.

[ZH15] ZHOU, LIANG and HANSEN, CHARLES D. "A survey of colormaps in visualization". *IEEE TVCG* 22.8 (2015) 2.