Quantitative Comparison of Dynamic Treemaps for Software Evolution Visualization

Eduardo Faccin Vernier[1,2]
Joao Comba
Institute of Informatics
[1]Federal University Rio Grande do Sul, Brazil
Email: comba@inf.ufrgs.br, efvernier@inf.ufrgs.br

Alexandru C. Telea
Johann Bernoulli Institute
[2]University of Groningen, the Netherlands
Email: a.c.telea@rug.nl

***Abstract*—Dynamic treemaps are one of the methods of choice for displaying large hierarchies that change over time, such as those encoding the structure of evolving software systems. While quality criteria (and algorithms that optimize for them) are known for static trees, far less has been studied for treemapping dynamic trees. We address this gap by proposing a methodology and associated quality metrics to measure the quality of dynamic treemaps for the specific use-case and context of software evolution visualization. We apply our methodology on a benchmark containing a wide range of real-world software repositories and 12 well-known treemap algorithms. Based on our findings, we discuss the observed advantages and limitations of various treemapping algorithms for visualizing software structure evolution, and propose ways for users to choose the most suitable treemap algorithm based on the targeted criteria of interest.**

I. Introduction

Hierarchies play a central role in understanding large software systems. Such systems evolve over hundreds of revisions or more, and can have thousands of elements or more, which are typically organized hierarchically (*e.g.* in folders, files, classes, and methods). Hence, tools for visually understanding evolving hierarchies are a key component in the program comprehension arsenal. Treemaps are a well known method for visualizing hierarchical data. Given an input tree whose leafs have several attributes, treemaps recursively partition a 2D spatial region into cells whose area, color, shading, or labels encode the tree's data attributes. Compared to other methods such as node-link [1], [2] or Sunburst [3], [4] techniques, treemaps use all available screen pixels to show data and thus can handle trees of tens of thousands of nodes.

Dynamic treemaps leverage the above advantages to show dynamic, or evolving, trees. Given a tree sequence, they create an animated sequence of treemap layouts that reflect how the structure and attributes of the trees in the sequence change in time. Evolving treemaps have been created both by using classical static treemap algorithms [5] or by specialized algorithms [6]–[8].

Evolving treemaps have received great interest in software visualization [7]–[11]. As many treemap techniques exist, the question emerged of how to measure their quality. For common rectangular treemaps, which map tree nodes to rectangles, visual quality is typically measured by the aspect ratio of these rectangles. However, the aspect ratio may not capture all desirable qualities of such treemaps. For example, bad aspect-ratio cells of a tiny area could influence the overall visual quality far less than large bad aspect-ratio cells. Atop visual quality, evolving treemaps are assessed by measuring their rate of visual change. However, this metric may not capture all desirable properties: Large visual changes in a treemap are expected (and actually desirable) when the underlying tree changes drastically, but undesired when the tree changes only slightly.

Although treemaps are used for over two decades in software visualization [5], [12]–[14], there are few comprehensive evaluations of the quality of dynamic treemap techniques and, to our knowledge, none that focuses on trees capturing software evolution. The aim of this paper is to fill this gap. For this, we first review the related work in (dynamic) treemaps and their quality measurement, with a focus on software visualization (Sec. II). We next refine desirable treemap properties into 5 quality metrics that capture both spatial quality and dynamic quality (Sec. III). We measure these metrics on 12 well-known treemap algorithms on 28 tree sequences, ranging from a few hundred to tens of thousands of elements, all extracted from software repositories. We next visualize and analyze our results to address questions that practitioners would like to answer to choose a suitable technique (Sec. IV). We discuss our findings and proposed methodology in Sec. V. Our results (datasets, metrics, treemap implementations, evaluation results, and visualizations thereof) are publicly accessible for researchers in the software visualization field interested in evaluating treemap methods for evolving software hierarchies.

II. Background

Hierarchies are arguably the central element in most software visualizations. They capture the physical (*e.g.* files and folders) or logical software (*e.g.* syntax tree) system structure, together with static or dynamic attributes, *e.g.*, code size, quality metrics [15], change requests, or testing results [9]. Both static and dynamic hierarchies in program comprehension are typically extracted by mining software repositories [16], [17]. When small (a few hundred nodes), such trees can be visualized using classical node-link layouts such as in class or architecture diagrams [16], [18], [19]. This works well for architecture-level views on a software system. However, code-level views, which contain nodes from subsystems all the way to classes and methods, generate large trees, having hundreds of thousands of nodes [4]. These require space-

filling methods, such as icicle plots [20], [21] or, the method of choice, treemaps. The latter are discussed below.

A. Treemap algorithms

Let $T = \{n_i\}$ be a tree with nodes n_i, and let $a_i \in \mathbb{R}^+$ be an attribute defined on the tree leaves. For non-leaf nodes n_i, a_i equals the sum of the attributes of the children of n_i. A rectangular treemap algorithm TM creates a set of rectangle cells $\{c_i\} = TM(T)$, $c_i \subset \mathbb{R}^2$ for the nodes n_i so that the area of c_i equals a_i and children node cells create a partition of their parent cell. Several treemap algorithms exist, as follows (for detailed surveys, see [5], [13], [14], [22]. Slice and dice (SND) treemaps pioneered the concept but were found to create too long-and-thin cells which are hard to grasp [12]. Subsequent algorithms tried to improve this aspect, quantified by the aspect ratio (AR) of the treemap cells. Squarified treemaps (SQR) propose a slicing heuristic that achieves, in general, very good (close to one) AR values [23]. Nagamochi and Abe refined this idea in an algorithm (APP) that approximates the optimal AR a given treemap can reach [24]. However, SQR is not particularly *stable* – small changes in the input tree can yield large changes in the treemap layout. Several algorithms have aimed to improve stability. Ordered treemaps (OT) [25] and Strip treemaps (STR) [26] layout cells c_i to follow a predefined order of the nodes n_i. Different algorithms propose different orderings: Pivot-by-Middle (PBM), Pivot-by-Size (PBZ), and Pivot-By-Split-Size (PBS) [25]; Engdahl's Split algorithm [27]; and laying out cells along a space-filling curve, *e.g.*, Spiral (SPI) [28], and Hilbert (HIL) and Moore (MOO) fractal curves [29]. Spatially-Ordered Treemaps (SOT) [30] extend SQR by ordering sibling nodes so that the most similar ones are processed in turn. NMap [31] uses a related idea; cells are placed according to the similarity of their attributes, using a dimensionality-reduction approach. Two versions exist: NMap Alternate Cuts (NAC) alternate horizontal and vertical cuts to subdivide the space (akin to SND), while NMap Equal Weights (NEW) splits the space to create similar-size cells of similar. However, NMap was only applied to single-level trees. Recently, Sondag *et al.* propose stable treemaps [6], which aim to improve both the AR and stability for dynamic treemaps by using non-sliceable layouts.

Other cell shapes can be used besides rectangles. Voronoi treemaps [32], [33] exploit the properties of weighted Voronoi diagrams to create organic-looking displays where cells are convex polygons with, in general, good AR values. Voronoi methods have also been used, with good results, to construct dynamic treemaps for visualizing software structure evolution [7], [11]. Hybrid treemaps (HTM) [34] combine various basic treemap techniques to generate the final layout. Other variants include jigsaw treemaps [35], orthoconvex treemaps [36], and bubble treemaps [37].

B. Treemap quality metrics

In practice, the quality of treemaps is measured using two types of metrics, as follows.

Spatial quality metrics capture how easy one can read the information shown in a static treemap. Such metrics include the aspect ratio (AR) of the treemap cells, which ideally should equal one. For ordered treemaps, the readability metric measures how often one switches visual scanning direction while reading the treemap in order [26]; and the continuity metric measures how often cells for neighbor nodes (following the given node order) are not neighbors in the treemap layout [28].

Stability metrics capture how easy one can follow the changes in a dynamic treemap. Given two treemaps for two (typically consecutive) time-moments t_i and t_j, Shneiderman and Wattenberg [25] define stability as the distance between the vectors $(x_k(t_i), y_k(t_i), w_k(t_i), h_k(t_i))$ and $(x_k(t_j), y_k(t_j), w_k(t_j), h_k(t_j))$, where x and y are the coordinates of the top-left corner, and w and h, the width, and the height of a cell c_k, averaged over all cells in the treemap. Hahn *et al.* [8] use for stability the change of distance between the centroids of $c_k(t_i)$ and $c_k(t_j)$, averaged over all cells. Tak and Cockburn [29] use the same cell-change metric (top-left corner, width, height) as Shneiderman and Wattenberg [25], but aggregate via variance rather than average. They also propose a drift metric which measures how much a cell moves away from its average position over a time period. Two recent metrics measure stability at the level of pairs of cells rather than individual cells. Hahn *et al.* [38] propose the relative direction change, which measures the angle change of centroids for every pair of cells in a layout. Sondag *et al.* [6] measure the relative position change of each cell with respect to eight planar zones defined by four lines given by the edges of that cell, averaged over all treemap cells.

C. Software visualization challenges

Summarizing, considerable effort went into designing static treemap methods and measuring their quality. Less effort went to evaluating dynamic treemaps. We identify limitations in several directions, with a focus on our use-case of visualizing large evolving software hierarchies:

Algorithms: Treemap papers typically compare a few (2..5) algorithms from the much larger set of available ones. In particular, it is not clear how most existing static treemap algorithms perform on the types of dynamic trees extracted from software evolution analyses.

Datasets: Existing methods are typically evaluated on one or a few datasets. While in this paper, we cannot (and do not aim to) cover the full space of all possible trees, we can do better than current work: For our specific context of software visualization, we aim to know how treemap methods perform on a representative collection of software hierarchies capturing software evolution.

Metrics: As outlined in Sec. II-B, stability is currently measured by looking at how much two treemaps (typically for consecutive time moments) do change with respect to each other. However, when the underlying tree sequence changes a lot, *e.g.* by insertions or deletions of many files or classes at the same moment during a software repository's evolution (an event well-known to take place often in software evolution), the treemap will change a lot, so its evolution will be labeled as unstable. However, it is actually *desirable* to have a large visual change in this case, as this correctly shows the presence

of a large data change. We argue that ways to measure stability as a function of the data change is needed.

Result exploration: Most evaluations consider only aggregated metrics with one value per technique or per technique-and-dataset. Analyzing the actual distribution of metric values over both layout-space and time can give extra insights into the strengths and weaknesses of specific techniques.

Replicability: Treemap evaluations can be hard to replicate as datasets and algorithm implementations are not always openly available or not integrated to make a comparison on different datasets, and along different metrics, easy. Replicability is a growing concern in information visualization but with particular weight in software visualization [39]–[41].

The remainder of this paper is dedicated to addressing the above points.

Dataset	Revisions	Nodes (total)	Average depth
animate.css	50	3454	2.87
AudioKit	22	11178	6.95
bdb	62	2658	3.83
beets	106	9844	3.75
brackets	88	120292	12.85
caffe	44	12969	4.93
calcuta	50	2882	10.76
cpython	321	584821	6.50
earthdata-search	46	18539	6.82
emcee	64	1746	3.62
exo	97	36436	11.88
fsharp	69	22906	7.89
gimp	72	170418	5.19
hospitalrun-frontend	38	16759	5.71
Hystrix	61	15530	13.29
iina	74	6849	4
jenkins	137	277185	11.94
Leaflet	84	13381	4.86
OptiKey	36	9782	6.72
osquery	37	14111	5.75
PhysicsJS	20	2022	4.6
pybuilder	53	5457	7
scikitlearn	88	48468	5.75
shellcheck	53	746	2.39
soundnode-app	35	3196	6.88
spacemacs	51	10201	4.96
standard	29	203	2
uws	122	4093	2.76
Totals:	2132	1458036	5.77

TABLE I
SOFTWARE EVOLUTION TREE DATASETS USED IN THE EVALUATION.

III. MEASURING THE QUALITY OF DYNAMIC TREEMAPS

To address the current limitations of dynamic treemap evaluations in software visualization, we performed an in-depth study covering the five directions in Sec. II-C, as follows.

A. Algorithms

We consider in our evaluation 12 methods: Approximate (APP), Hilbert (HIL), Moore (MOO), NMap-Alternate-Cuts (NAC), NMap-Equal-Weights (NEW), Pivot-by-Middle (PBM), Pivot-by-Size (PBZ), Pivot-by-Split-Size (PBS), Slice-and-Dice (SND), Spiral (SPI), Squarified (SQR), and Strip (STR) treemaps. For NMap, we use as seed layout the one computed by SQR (for details, see [31]). We do not consider non-rectangular treemap methods, as their quality is less easy to compare with rectangular ones, and are also less used in practice. Also, we do not consider the stable treemaps in [6] as this method is considerably slower (over one order of magnitude) than the above-mentioned methods.

B. Datasets

We evaluate all above treemap methods on a collection of 28 datasets (Tab. I). All of them consist of trees describing the hierarchy of public and well-known GitHub software repositories (folders, files, classes), one tree per revision, where leaves (classes) are attributed by their number of lines of code. The trees and their attributes have been extracted from the actual repositories by a fully automatic pipeline we built using *libgit2* [42] for repository parsing and Understand [43] for code analysis. For a more detailed description of the extraction pipeline, we refer to [44]. The respective software projects have widely different sizes, tree depths and structures, durations, numbers of contributors, language (C, C++, Java, Python), and code type (library, framework, application). This is seen in the figures in Tab. I and also in Fig. 1 which shows the union trees $\cup_i T(t_i)$ for the considered datasets. Hence, we argue that this collection covers reasonably well the space of tree sequences obtained from software evolution.

C. Metrics

Let w_k and h_k be the weight and height of cell c_k; and (W,H) the width and height of the screen space we draw the treemap in. With these, we consider the following metrics.

1) Spatial quality metric: We first consider the classical aspect-ratio metric

$$Q_k^{AR} = \min(w_k,h_k)/\max(w_k,h_k). \quad (1)$$

Introduced in [23], this metric has been since then used all treemap evaluations to capture spatial quality. As such, we keep it in our evaluation. It is designed to give high scores for rectangles with sides of similar length, and low scores otherwise.

2) Stability metrics: Let $c_k(t_i)$ and $c_k(t_j)$ be two cells in two consecutive versions $T(t_i)$ and $T(t_j = t_{i+1})$ for the same node in a dynamic tree. Typical stability metrics (Sec. II-B) only measure the *visual* change δc_k between $c_k(t_i)$ and $c_k(t_j)$. We use for δc_k the average sum of distances between the four corresponding corners of $c_k(t_i)$ and $c_k(t_j)$ [25], normalized by the treemap diagonal $\sqrt{W^2+H^2}$, so $\delta \in [0,1]$. We next define the *data change* between nodes $n_k(t_i)$ and $n_k(t_j)$ as $\delta a_k = |a_k(t_i) - a_k(t_j)|$, where a_k is the relative weight of n_k at time t_i. If either of $n_k(t_i)$ or $n_k(t_j)$ does not exist, *i.e.*, a node was created or deleted in versions t_i or t_j, we set the respective a_k to zero, which is as if the respective node was depicted by a zero-size cell. We normalize $a_k(t_i)$ by the weight sum of all nodes n_k present at time t_i, so $\delta a_k \in [0,1]$. With this, we define the stability of a cell c_k in a treemap in several ways. First, we define stability as

$$Q_k^{RATIO} = (1-\delta c_k)/(1-\delta a_k). \quad (2)$$

When visual changes are proportional to data changes, since both are normalized, Q_k^{RATIO} goes to one. Note that an analogy to Eqn. 1, *i.e.*, $Q_k^{RATIO} = \min(\delta c_k, \delta a_k)/\max(\delta c_k, \delta a_k)$ does not work: Eqn. 1 is symmetric in width and height. For stability

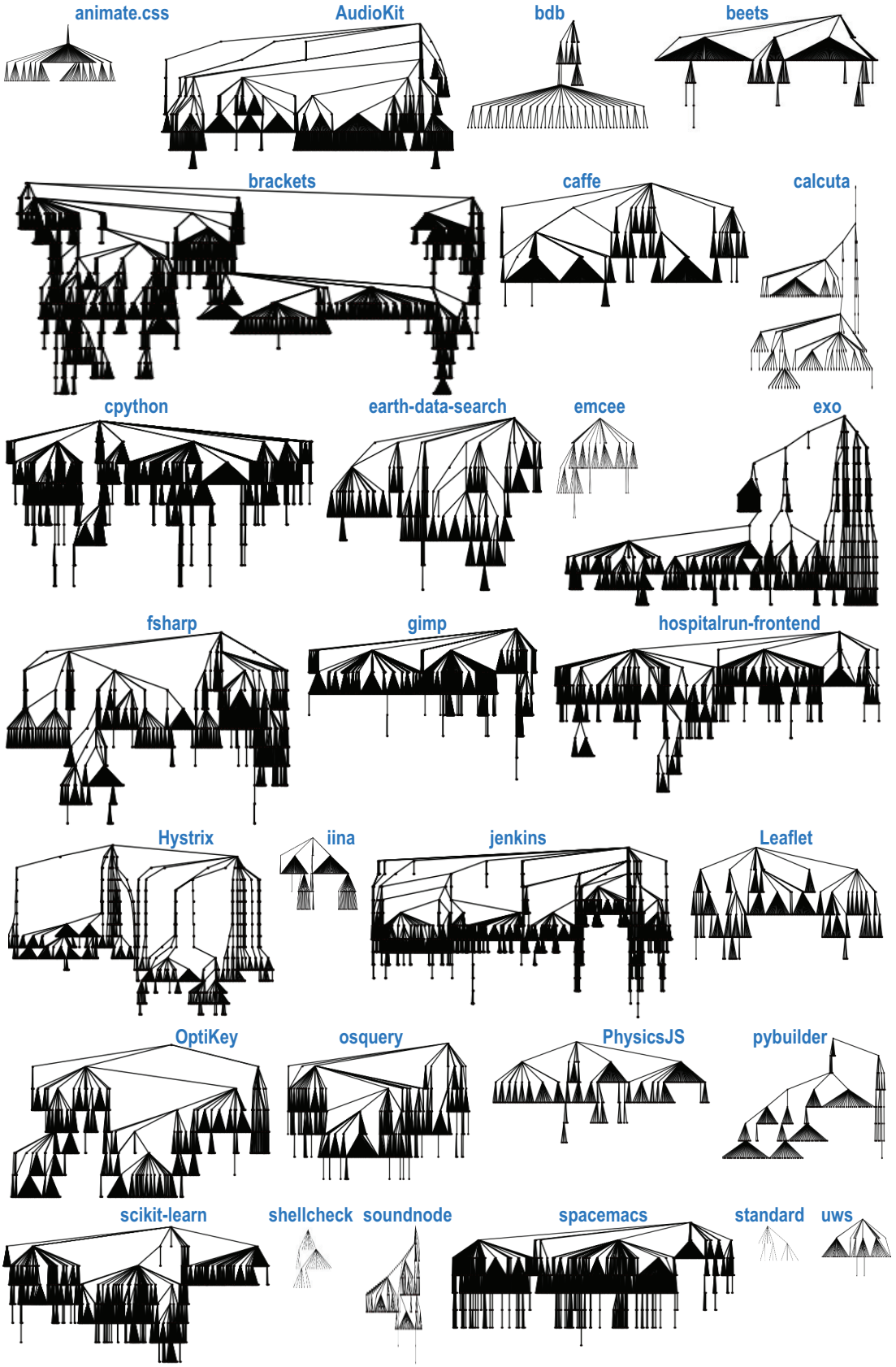

Fig. 1. Union trees of software evolution tree datasets used in the evaluation. Names correspond to public repositories on GitHub.

(Eqn. 2), we want to assess visual change as a function of data change, and not conversely.

A second way to define stability is by

$$Q_k^{MOD} = 1 - |\delta c_k - \delta a_k|. \quad (3)$$

For proportional visual *vs* data changes, $Q_k^{MOD} = 1$.

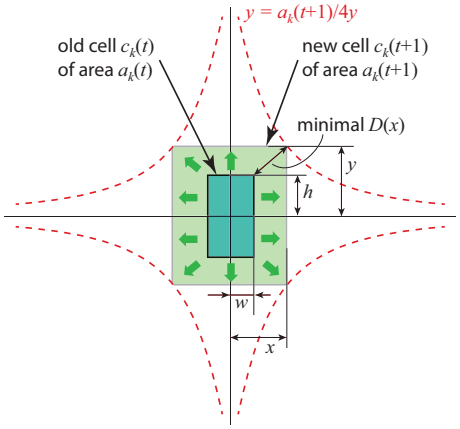

Fig. 2. Computation of unavoidable change metric Q_k^{UNAV}.

To compare data and visual changes, Q_k^{RATIO} and Q_k^{MOD} must be normalized to the same range, therefore we clip Q_k^{RATIO} to the $[0,1]$ interval. However, this can introduce normalization biases, *e.g.* when the data changes and visual changes have very different ranges. To address this, we next propose to define stability purely in visual space. For this, we consider the actual change δc_k of a cell *vs* the *unavoidable*, *i.e.* minimal, change Δc_k that c_k would need to undergo to accommodate the data change from $a_k(t)$ to $a_k(t+1)$. If $\delta c_k > \Delta c_k$, the algorithm is unstable; if $\delta c_k = \Delta c_k$, it is fully stable. We compute Δc_k as follows (Fig. 2). Let $c_k(t)$ be a cell of width w and height h at time step t. Let $c_k(t+1)$ be the version of $c_k(t)$, of area $a_k(t+1)$, at step $t+1$. We first note that δc_k is minimal when $c_k(t)$ and $c_k(t+1)$ have the same center, as visual change is then caused *purely* by data change and not by avoidable 'drift' of the cell corners. Taking a xy coordinate frame centered in this common cell center, the top-right corner of $c_k(t)$ is constrained to a hyperbola $y = a_k(t+1)/4x$. Hence the minimal change Δc_k is four times the minimal distance D from this corner to the hyperbola, *i.e.*

$$D(x) = \sqrt{(x - w/2)^2 + (a_k(t+1)/4x - h/2)^2}.$$

To find the minimum of D, we solve $\frac{dD^2}{dx} = 0$ for $x \geq 0$. This quartic equation in x has analytic solutions. We obtain x, the width of the optimal cell $c_k(t+1)$, and thereby the minimal Δc_k. Finally, we define the unavoidable-motion stability as

$$Q_k^{UNAV} = 1 - (\delta c_k - \Delta c_k). \quad (4)$$

Finally, we define stability for a whole tree T as the *absolute* value of the Pearson correlation coefficient

$$Q^{CORR} = \left| \frac{\sum_k (\delta c_k - \overline{\delta c_k})(\delta a_k - \overline{\delta a_k})}{\sqrt{\sum_k (\delta c_k - \overline{\delta c_k})^2} \sqrt{\sum_k (\delta a_k - \overline{\delta a_k})^2}} \right| \quad (5)$$

of the signals $\{\delta c_k\}$ and $\{\delta a_k\}$ for all cells $c_k \in T$, where $\overline{\delta c_k}$ and $\overline{\delta a_k}$ are the signals' averages, so $Q^{CORR} \in [0,1]$. If visual and data changes δc_k and δa_k are linearly correlated, Q^{CORR} reaches one. Q^{CORR} close to zero indicates uncorrelated changes, *i.e.*, instability.

Compared to existing treemap stability metrics [6], [8], [25], [29], all our above metrics consider the *relation* of visual change δc_k to data change δa_k. This is a fundamental difference: A treemap method $TM(T) = \{c_i\}$ is a *function* from trees T to cell-sets $\{c_i\}$, so its stability should be defined akin to Cauchy or Lipschitz continuity, which relate function-value ($\{c_i\}$) changes to variable (T) changes rather than measuring function changes only. Indeed: If a function strongly changes, the function *itself* is not necessarily unstable; this can happen when the input variable strongly changes.

3) Metric weighting: As mentioned in Sec. I, very small but bad aspect-ratio cells may not strongly influence the overall perceived spatial quality of a treemap, since they are barely visible. The same argument could be made for very small unstable cells *vs* the overall perceived stability. To model these, when computing the average value of the metrics Q^{AR}, Q^{RATIO}, Q^{MOD}, and Q^{UNAV}, we weigh the respective per-cell values Q_k^{AR} (and the other three ones) by the sizes a_k of their cells. We used such weighted metrics in all experiments described next in Secs. IV-B-IV-D. However, the obtained results showed that the aggregated weighted metric values differ only very slightly from their unweighted versions. As such, in the following we will only consider the unweighted metric versions.

IV. Result Exploration

We measure the five metrics (Eqns. 2-5) on all 28 test datasets (Sec. III-B) processed by all 12 treemap methods (Sec. III-A). We record metrics at the *cell* level (except Q^{CORR}, recorded at tree level). This yields a high-dimensional-and-hierarchical dataset, conceptually a table with seven columns (5 metrics, algorithm ID, dataset ID, time step) and as many rows as the number of measured cells in all datasets, all timesteps. Exploring this data space is a challenge in itself. As noted in Sec. II-C, current treemap evaluations typically present only a few metrics, aggregated to a single (typically average) value per algorithm or per algorithm-and-dataset. To get more insight, we propose several visualizations that present various aspects of the evaluation data to answer specific questions concerning the evaluated algorithms. We proceed in a bottom-up fashion: We first explore the data at the finest (cell) level-of-detail (Sec. IV-A). This shows subtle differences between different methods (we show all table rows), but cannot show all evaluated metrics (table columns). Next, we study the quality as a function of time, for one given evolution sequence (Sec. IV-B). Thirdly, we compare the aggregated 5 metrics for all dataset and algorithm combinations (Sec. IV-C). Finally, we aggregate all results to present a compact comparison of all algorithms (Sec. IV-D).

A. How does visual change relate to data change (Q1)?

Before actually evaluating stability, we want to study the distribution of visual changes created by the tested algorithms as function of the respective data changes for all datasets,

all timesteps. For this, we show a scatterplot per algorithm (Fig. 3), where, for all datasets, x maps $\delta a_i(t_j)$, *i.e.* data change of all cells c_i from time step t_j to t_{j+1}, for all time steps j; and y maps $\delta c_i(t_j)$ (see Sec. III-C2). A point is thus a cell in a revision of a dataset. To account for overplotting, we compute density maps from these scatterplots using kernel density estimation [45] and color-code the density using a heat colormap. Ideally, the visual change should be proportional to

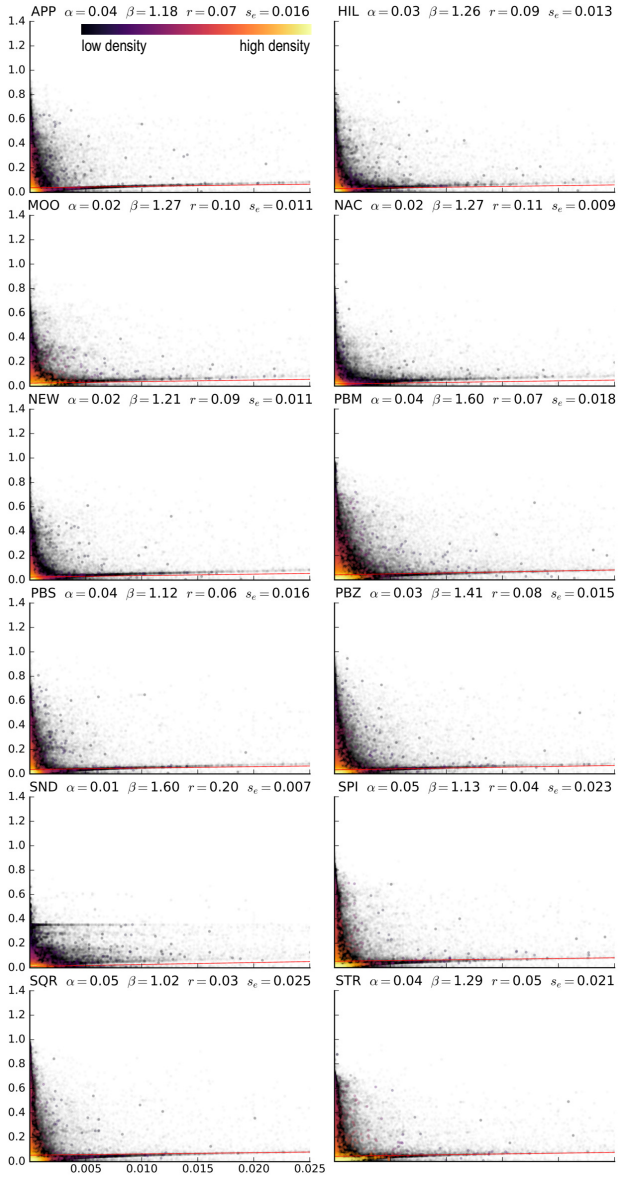

Fig. 3. Correlation of data and visual change per algorithm, all datasets.

data change (Sec. III-C2), so our scatterplots should be close to a diagonal line. We see that this is not the case. All plots show an upwards-pointing 'tail' close to the origin. This tells that most cells with small data changes have disproportionately large visual changes, so instability affects more the small than the large cells. Shallower tails indicate more stable methods, *e.g.* SND. To get a more summarized insight, we also plot a linear-regression line (red), characterized by the slope (α) and the y-intercept (β), and compute the linear correlation coefficient (r) and standard error (s_e) of the points. Larger r coupled with small s_e values indicate methods which correlate visual change with data change better, *e.g.* SND and NAC. We also find the worst-correlating methods, SQR and PBS, and see that SQR is about 7 times worse than SND.

B. How is quality evolving in time (Q2)?

Q1 does not show how quality fluctuates over time for a given tree sequence. Knowing this is important to assess what one can expect when using a given treemap algorithm for a sequence of hundreds of revisions extracted from a repository. To assess this, we show a chart per method, per dataset, and per metric family (that is, spatial quality Q^{AR} and per-timestep averaged values of the four stability metrics Q^{RATIO}, Q^{MOD}, and Q^{UNAV}). In all charts, x maps time and y shows a box plot indicating median (black), 25-75% range (green), and 5-95% range (gray). Since we cannot show this chart for all our 28 datasets (nor can we aggregate them in a single chart), we select one representative dataset to depict: *cpython*. The dataset was extracted from the official Github repository hosting the source code of the Python programming language [46]. This is our largest dataset with 321 revisions and an average of over two thousands tree nodes per revision. Results for other datasets can be found online [47].

Figure 4a shows the evolution of the Q^{AR} metric (Eqn. 1) for all tested methods for *cpython*. We see that APP and PBS deliver overall quite high and constant-over-time aspect ratios (0.7), so they are the best methods for spatial quality, with APP being better as it has a narrower Q^{AR} spread around a slightly higher median value. SQR scores higher median values, but has a larger spread – for every revision, it can score as bad as 0.05 aspect-ratio, while APP does not drop below 0.4 (compare the bottoms of the gray bands in Fig. 4 for APP and SQR). SND shows the worst spatial quality, with a tight spread around a median Q^{AR} below 0.1. The chart also tells us that most methods deliver *consistent* spatial quality regardless of the data changes in the 321 revisions (which we found to be large by manually examining the sequence). The quality decrease shown by SND and (less) by HIL and STR is somehow surprising, as none of the studied methods uses a 'history' of the tree-sequence in its layout heuristics.

Figure 4b shows the evolution of four stability metrics Q^{RATIO}, Q^{MOD}, and Q^{UNAV}, averaged per time-step. Compared to spatial quality, we see now much more variation between methods and also much more variation (of the stability) over time. We see that SND is by far the most stable method, whereas SQR, SPI, and PBM score worst. Long 'icicle' like boxplots indicate revisions where much more visual change was present than 'warranted' by the data change. Interestingly, these appear at the same moments for different algorithms (Fig. 4b, red markers shows one example). For such moments we see large variations across methods: For SPI, this is the most unstable part of the sequence, both in median and 5-95% range sense, whereas APP finds earlier sequences (marked in blue in Fig. 4) which are harder to lay out stably.

C. How do methods perform on different datasets (Q3)?

So far, we presented charts aggregate over all datasets (Sec. IV-A) or focus on a single dataset, but aggregate all

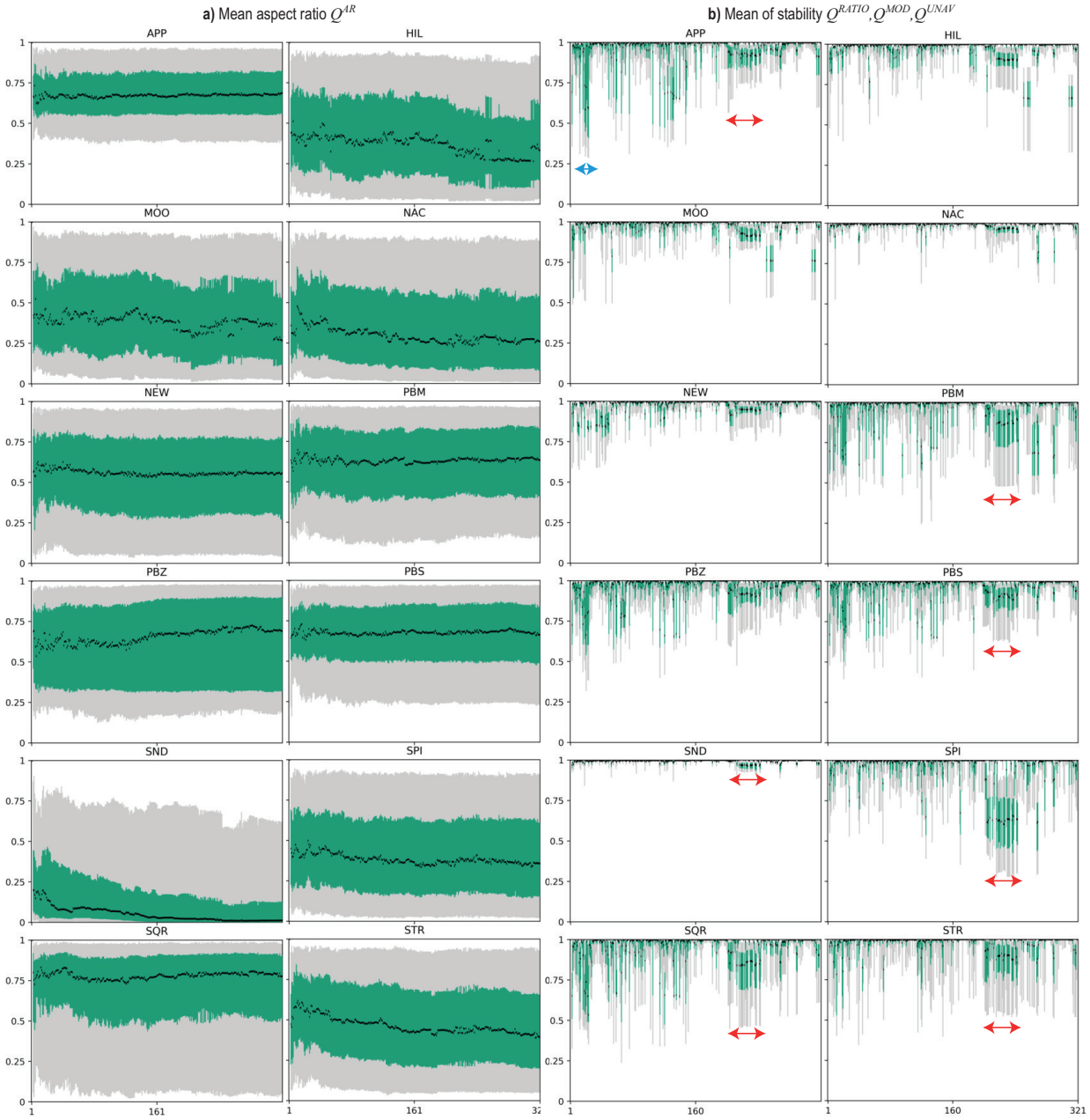

Fig. 4. Evolution in time of spatial quality (a) and averaged four stability metrics (b) for the *cpython* dataset.

stability metrics (Sec. IV-B). We would like to see how the proposed stability metrics compare to each other, as we are still in the process of understanding their measurement characteristics. Also, we would like to see how these metrics vary over several datasets. For these goals, we use a set of table views, one per quality metric. In each table, columns are datasets and rows are algorithms, respectively. Each cell thus encodes the average value of one quality metric for one dataset tested by one algorithm. Cells are colored with a luminance-based colormap, with data values separately normalized per metric table, so that darkest cells indicate worst cases in all tables (but with potentially different metric absolute values), and brightest cells indicate best cases in all tables, respectively.

Figure 5 tells several interesting things. Scanning the first table row-wise, we see that there are no large aspect-ratio quality differences between the tested datasets. This tells that most methods (with the notable exception of SND) achieve quite good aspect ratios for a wide dataset variation. Over all datasets, APP is the best method, surpassed by SQR only for a few datasets. Conversely, we see that SND is the most stable method with respect to all four considered stability metrics. Stability-wise, we see that some datasets (*hospitalrun-frontend*, *Leaflet*, and *PhysicsJS*) consistently score worse than all others for basically all algorithms. These are also the datasets yielding the worst stabilities, when PBM, SQR, STR, and SPI methods are used. This indicates that these methods

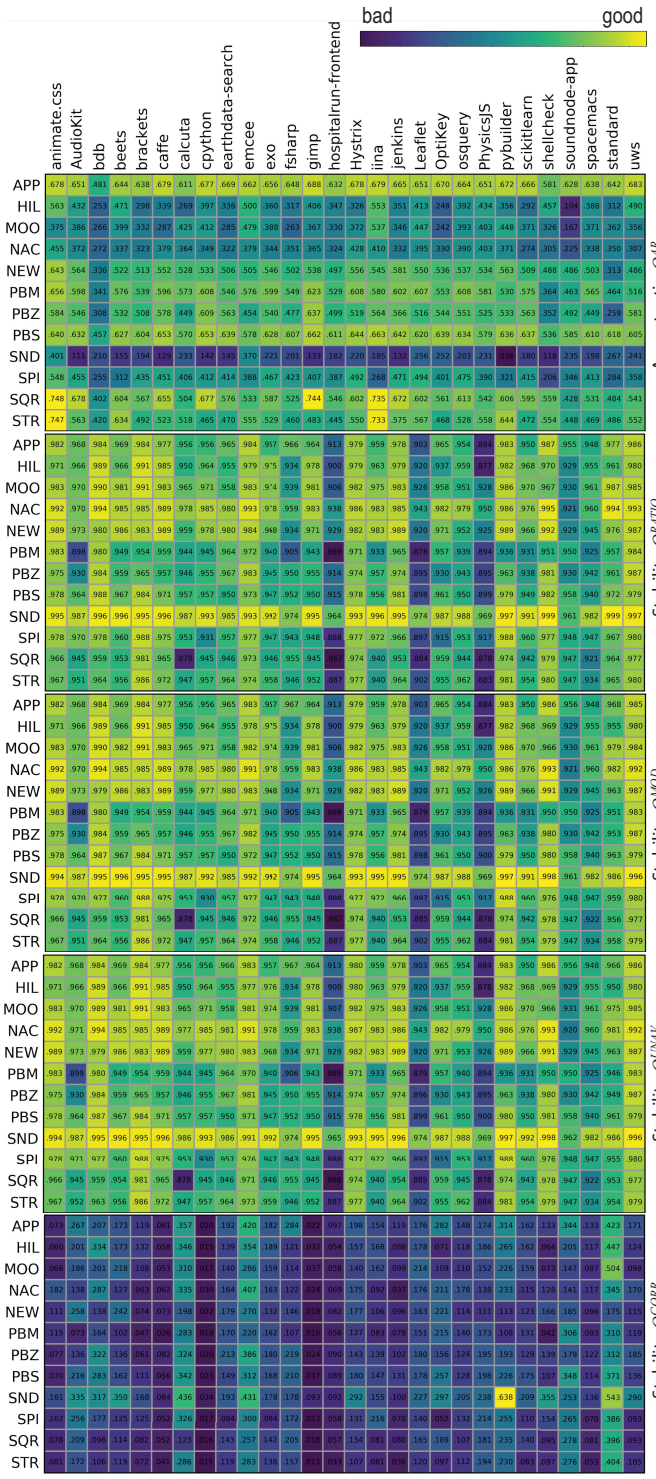

Fig. 5. The five quality metrics for all tested methods, all datasets.

are quite sensitive in stability on the type of input dataset so, for obtaining higher stabilities, other methods should be used. At a higher level, we see that the Q^{RATIO}, Q^{MOD}, and Q^{UNAV} stability metrics yield very similar plots. This is an interesting findings, since the metrics have quite different formulations (Sec. III), and indicates that the results can be trusted – the chance of three metrics having such different expressions yielding so similar values being very small. In contrast, the Q^{CORR} metric has much lower values, which is explained by the fact it is much more conservative – a good algorithm would need to yield very well correlated δa_k and δc_k values, and we have seen in Sec. IV-A that this is by far not the case. We conclude that visual *vs* data change correlation is a too strong quality desiderate for dynamic treemaps handing large real-world datasets, and advise next to use in practice any of the Q^{RATIO}, Q^{MOD}, and Q^{UNAV} metrics to gauge stability, or, as we have done in Sec. IV-C, their average value.

D. How to summarize the comparison (Q4)?

The visualizations so far (Q1..Q3) have given us several insights: We have seen that APP, PBS, and SQR are the best methods with respect to spatial quality, while SND performs poorly for that, but it is the best for stability; different methods have quite different spreads of quality over a given tree sequence, some delivering more consistent results than others, but for most algorithms do not degrade over time; and several of the proposed stability metrics are strongly correlated. It is now useful to *summarize* our findings to present a compact ranking of the tested methods. For this, we use two stacked bar charts. Each bar maps one method and is divided into segments. A segment's length tells the percent of the total number of versions (of all datasets) for which that method had a specific rank regarding spatial quality (Fig. 6a) and averaged stability metrics (Fig. 6b). We color segments by an ordinal colormap to show these ranks (1 being the best and 12 being the worst). Bars (methods) are sorted in each chart to put the one with highest average rank, weighted by the percents of the total number of versions for all obtained ranks, at the top (Fig. 6). From Fig. 6, we first see that spatial quality and stability are strongly inversely correlated – methods that score well on one tend to score poorly on the other. We also see that the top methods in both charts are very good for *most* of the tested datasets, *i.e.*, it is easy to find a method that optimizes either spatial quality or stability, but not both. Interestingly, APP (a less known method) is better in spatial quality, and significantly better in stability, than SQR (arguably the method of choice for creating good aspect-ratio treemaps), so it should be preferred to SQR. Similarly, for stability, APP and NEW (two less known methods) are in the top-four most stable methods, and while worse than SND (very well known method), they have higher spatial quality, so they should be preferred to SND.

A disadvantage of the rank charts in Fig. 6 is that they do not easily allow linking spatial quality and stability. To alleviate this, we propose a final visualization which uses a start plot metaphor (Fig. 7). The scatterplot points (circles, categorically colored) are methods attributed by their average spatial quality and stability over all datasets, all revisions. Each method

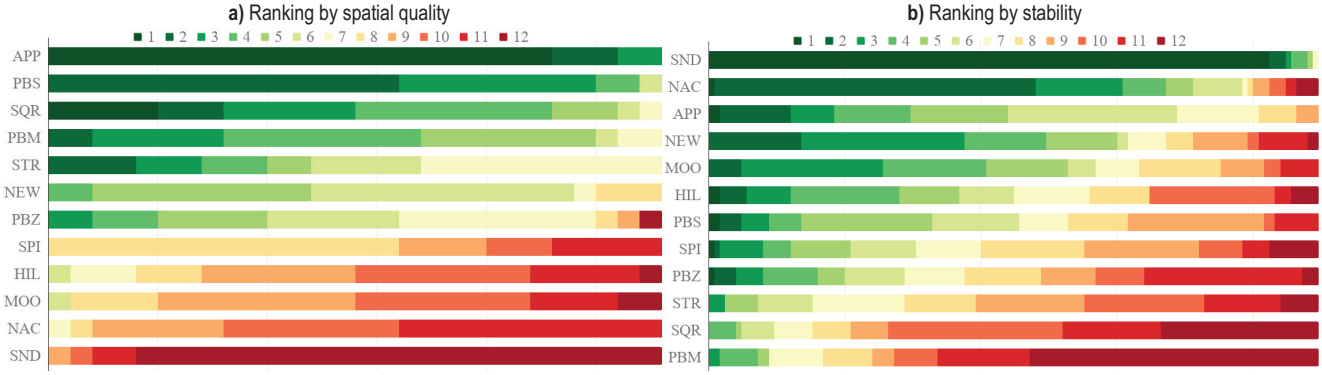

Fig. 6. Ranking of the 12 methods showing the percentage of times they scored a certain rank with respect to spatial quality (a) and averaged stability (b).

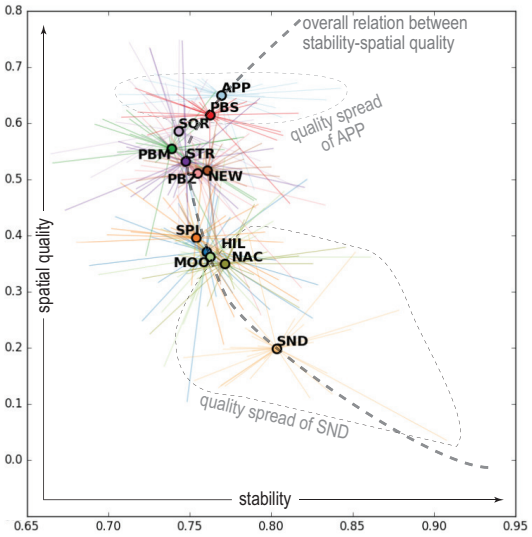

Fig. 7. Summarized comparison of all methods (colored dots) on all datasets (colored lines) *vs* spatial quality and stability.

is linked with the 28 tested datasets by same-color lines; a line's endpoint has the average spatial quality and stability over all its revisions for the corresponding method. The plot conveys several insights: First, methods follow roughly a concave curve (Fig. 7, thick dashed curve), telling the trade-off between spatial quality and stability. Variation in average spatial quality is much larger (roughly 45%) than in average stability (roughly 8%). The fan-out of lines from a method shows how predictable that method is, and here we see large variation over methods, with *e.g.* APP being quite consistent in spatial quality, while MOO, STR, and SND show large dataset-dependent variations in both spatial quality and stability (see Fig. 7, thin dashed curves). The latter is especially interesting: Even though SND has the highest average stability, it can also score worse than many other methods on certain datasets.

To conclude, it is hard to designate an 'optimal' method, as this strongly depends on which of stability and spatial quality users see as most important for their concrete use-cases, and by how much. Still, based on all our insights, we believe that APP offers a very good compromise – very high spatial quality and overall stability similar to most methods, surpassed only (and not in all cases) by SND.

V. Discussion

Let us discuss our results in the light of the dimensions of evaluating treemap algorithms for software evolution visualization (Sec. II-C):

Algorithms: We consider 12 well-known treemap methods, in contrast to typically 2..3 techniques in current treemap evaluations in software visualization papers. We argue that this gives valuable insights on the suitability of such well-known methods for handling evolving software trees, so it makes the choice of a given method easier for the software visualization practitioner. For instance, our evaluation can tell the interested user which are the advantages (or limitations) of a *given* algorithm *vs* another given algorithm, from the perspectives of spatial quality or stability.

Datasets: Our treemap benchmark cannot cover all variations of trees extracted from software evolution use-cases. However, it measures in total roughly $1.9 \cdot 10^9$ treemap cells for 28 tree sequences up to 321 time-steps (revisions). The size and variability of tree sequences covered by our study is larger than all existing similar evaluations of dynamic treemaps in software visualization. However, we admit that we cannot *extrapolate* from this evaluation to draw statistically strong conclusions concerning the quality of a given treemap algorithm for the *entire* space of evolving software hierarchies. Doing so would require (a) a characterization of this space in terms of objective metrics (*e.g.*, tree size, depth, type of structure, type of changes); (b) a targeted search of software repositories to extract trees which 'sample' well all these dimensions; (c) an evaluation of our metrics on this benchmark; and (d) most importantly, finding possible correlations between the measured performance of algorithms and the characteristics of the tree sequences they work on. We acknowledge these limitations, and outline them as important directions for future work.

Metrics: We measure treemap stability by essentially considering the first derivative of the treemap algorithm function mapping from tree-node weights to rectangular cell-sets. We detail four variants for measuring stability this way, and observe that three of them, while quite different in terms of actual definitions, yield very similar results. We believe this is an important finding, as it motivates the idea of defining stability by relating visual change to data change. The fourth stability metric (Pearson correlation) showed however to be of limited

practical use, as typical dynamic treemaps exhibit a too low correlation of the data and visual changes as compared to other phenomena where this metric is used. This can also indicate that dynamic treemaps may exhibit a more *complex* form of data *vs* visual change correlation than *linear* one. Concluding, we argue that measuring stability by involving both visual and data change is desirable, but we acknowledge that more work is needed to further refine the definition of the proposed stability metric, so that it avoids potential normalization biases, and it also captures in a more demonstrable way what actual users perceive as 'unstable'.

Result exploration: We present five visualizations of treemap quality metrics, covering all involved dimensions: cells, revisions, datasets, metrics, and algorithms. As the dimensionality of this data space is large, we obviously cannot cover *all* possible viewpoints. Yet, our visualizations help finding novel insights on the behavior of dynamic treemaps for evolving software hierarchies, and also confirm earlier observations, *e.g.* the known stability of SND. Our visualizations can be used to both analyze fine-grained details (at cell level) and present aggregated conclusions (at algorithm level). They can help the practitioner in understanding what is gained, and/or lost, by choosing a certain treemap algorithm instead of another one.

Replicability: All our results (datasets, treemap and visualization code, measurements) are available online at [47]. To our knowledge, this is the first benchmark for (dynamic) treemaps for applications in software evolution understanding. It can serve both for practitioners interested in choosing an algorithm based on specific quality criteria, but also for researchers aiming to benchmark their new algorithms, with limited effort, against existing ones.

Limitations: There are several points which can be better covered better. First and foremost, as mentioned, we need a more principled sampling of the space of trees extracted from software evolution to gain more confidence in the obtained quality results (or how these would differ as a function of the tree sequences' characteristics). We argue that our current work, *i.e.* the *automated* set-up of the extraction pipeline of dynamic trees from software repositories, computation of the proposed quality metrics, and visualizations that aggregate these, forms the necessary basis for such extensions, which we consider as future work. Separately, more treemap algorithms could be considered, *e.g.*, Voronoi, hybrid, or bubble ones. This will require an adaptation of the spatial quality and stability metrics so they can be used for non-rectangular cells.

Threads to validity: Similar to software quality, we measure treemap quality by a number of 'proxy' metrics. While we argue for these metrics at technical level (see the stability metric *vs* function continuity discussion) or, separately, reuse well-known metrics (see the aspect-ratio metric), we do not have hard evidence that such metrics truly capture quality as seen by the eyes of the beholder (end user). The advantage of using such 'intrinsic' quality metrics is that they can be computed automatically, on a large benchmark, and are independent on actual tasks, users, or use-cases. This allows for direct and objective comparisons, parallel to what is done on the context of *e.g.* Graph Drawing [48], [49], where metrics such as number of crossings, angle of crossings, and distribution of edge lengths are used to rank the quality of graph drawing algorithms. The disadvantage is that we cannot directly infer, from such metrics, how fit to purpose a given treemap technique will be given a specific user, use-case, type of dataset, and task. We argue for our approach as follows: *if* for a given user, use-case, and task, one agrees that a good treemap algorithm should have the properties captured by our quality metrics, *then* one can use our evaluation and related artifacts (benchmark, metrics, visualizations) to find the best suitable algorithms.

VI. Conclusions

We have presented an evaluation of treemap algorithms for the visualizaton of dynamic tree sequences extracted from the evolution of software repositories. For this, we proposed a benchmark formed by 28 datasets extracted from well-known such software repositories, five metrics that aim to capture spatial quality and stability, and 12 known treemap methods. We also propose six visualizations aimed at interpreting the measurement results from several angles, covered by four types of questions. All results (datasets, treemap implementations, measurement code, and visualizations) are publicly available and can constitute the basis of a benchmark for treemap evaluation for visualizing evolving software hierarchies.

Several directions exist for extending this work. First and foremost, a finer-grained analysis of the space of evolving software trees can be made to elicit correlations between characteristics of the datasets and measured quality of the tested treemap methods. Secondly, and at a higher level, it would be useful to extend this type of benchmarking to other application domains that generate dynamic trees and use treemap methods to visualize their evolution in time.

References

[1] D. Harel and Y. Koren, "Graph drawing by high-dimensional embedding," in *Revised Papers from the 10th International Symposium on Graph Drawing*, ser. GD '02. Springer-Verlag, 2002, pp. 207–219.

[2] A. Frick, A. Ludwig, and H. Mehldau, "A fast adaptive layout algorithm for undirected graphs," in *Proceedings of the DIMACS International Workshop on Graph Drawing*, ser. GD '94. Springer-Verlag, 1995, pp. 388–403.

[3] J. Clark, "Multi-level pie charts," 2006, https://neoformix.com/2006/MultiLevelPieChart.html.

[4] A. Telea, H. Hoogendorp, O. Ersoy, and D. Reniers, "Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study," in *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, Sept 2009, pp. 81–88.

[5] H.-J. Schulz, "Treevis.net: A tree visualization reference," *IEEE CG&A*, vol. 31, no. 6, pp. 11–15, 2011.

[6] M. Sondag, B. Speckmann, and K. Verbeek, "Stable treemaps via local moves," *IEEE TVCG*, 2017.

[7] R. van Hees and J. Hage, "Stable and predictable Voronoi treemaps for software quality monitoring," *Inf Soft Technol*, vol. 87, no. C, pp. 242–258, 2017.

[8] S. Hahn, J. Trümper, D. Moritz, and J. Döllner, "Visualization of varying hierarchies by stable layout of Voronoi treemaps," in *Proc. IEEE IVAPP*, 2014, pp. 50–58.

[9] S. Diehl, *Software Visualization – Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[10] D. Fisher and A. Sud, "Animated, dynamic Voronoi treemaps," in *Proc. EuroVis – posters*. Eurographics, 2010.

[11] D. Gotz, "Dynamic Voronoi treemaps: A visualization technique for time-varying hierarchical data," *Computer Science – Research and Development*, vol. 18, pp. 132–141, 2011, also as IBM Research Report RC25132 (W1103-173).

[12] B. Shneiderman, "Tree visualization with tree-maps: 2-D space-filling approach," *ACM TOG*, vol. 11, no. 92, 1992.

[13] H.-J. Schulz, S. Hadlak, and H. Schumann, "The design space of implicit hierarchy visualization: A survey," *IEEE TVCG*, vol. 17, no. 4, pp. 393–411, 2011.

[14] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. J. van Wijk, J.-D. Fekete, and D. W. Fellner, "Visual analysis of large graphs: State-of-the-art and future research challenges," *CGF*, vol. 30, no. 6, pp. 1719–1749, 2011.

[15] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer, 2006.

[16] M. Lanza and S. Ducasse, "Polymetric views – a lightweight visual approach to reverse engineering," *IEEE Trans Soft Eng*, vol. 29, no. 9, pp. 782–795, 2003.

[17] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Software: Evolution and Process*, vol. 19, no. 2, pp. 77–131, 2003.

[18] H. A. Müller and K. Klashinsky, "Rigi – a system for programming-in-the-large," in *Proc. IEEE ICSE*, 1988, pp. 80–86.

[19] A. Telea, A. Maccari, and C. Riva, "An open toolkit for prototyping reverse engineering visualizations," in *Proc. Data Visualisation (VisSym)*, 2002, pp. 241–249.

[20] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE TVCG*, vol. 12, no. 5, pp. 741–748, 2006.

[21] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen, "Understanding execution traces using massive sequence and circular bundle views," in *Proc. IEEE ICPC*, 2007, pp. 271–280.

[22] B. Shneiderman and C. Plaisant, "Treemaps for space-constrained visualization of hierarchies," 2017, https://cs.umd.edu/hcil/treemap-history.

[23] M. Bruls, K. Huizing, , and J. J. V. Wijk, "Squarified treemaps," in *Proc. VisSym*. Springer, 2000, pp. 33–42.

[24] H. Nagamochi and Y. Abe, "An approximation algorithm for dissecting a rectangle into rectangles with specified areas," *Discrete Applied Mathematics*, vol. 155, no. 4, pp. 523–537, 2007.

[25] B. Shneiderman and M. Wattenberg, "Ordered treemap layouts," in *Proc. IEEE InfoVis*, 2001, pp. 73–80.

[26] B. Bederson, B. Shneiderman, and M. Wattenberg, "Ordered and quantum treemaps: Making effective use of 2D space to display hierarchies," *ACM TOG*, vol. 21, no. 4, pp. 833–854, 2002.

[27] B. Engdahl, "Ordered and unordered treemap algorithms and their applications on handheld devices," 2005, MSc thesis, Dept. of CS, Stockholm Royal Institute of Technology.

[28] Y. Tu and H.-W. Shen, "Visualizing changes of hierarchical data using treemaps," *IEEE TVCG*, vol. 13, no. 6, pp. 1286–1293, 2007.

[29] S. Tak and A. Cockburn, "Enhanced spatial stability with Hilbert and Moore treemaps," *IEEE TVCG*, vol. 19, no. 1, pp. 141–148, 2013.

[30] J. Wood and J. Dykes, "Spatially ordered treemaps," *IEEE TVCG*, vol. 14, no. 6, pp. 1348–1355, 2008.

[31] S. Duarte, F. Sikanski, F. Fatore, S. Fadel, and F. Paulovich, "Nmap: A novel neighborhood preservation space-filling algorithm," *IEEE TVCG*, vol. 20, no. 12, pp. 2063–2071, 2014.

[32] M. Balzer and O. Deussen, "Voronoi treemaps," in *Proc. IEEE InfoVis*, 2005, pp. 49–56.

[33] M. Balzer, O. Deussen, and C. Lewerentz, "Voronoi treemaps for the visualization of software metrics," in *Proc. ACM SOFTVIS*, 2005, pp. 165–172.

[34] S. Hahn and J. Döllner, "Hybrid-treemap layouting," in *Proc. EuroVis (short papers)*, 2017.

[35] M. Wattenberg, "A note on space-filling visualizations and space-filling curves," in *Proc. IEEE InfoVis*, 2005, pp. 181–186.

[36] M. D. Berg, B. Speckmann, and V. van der Weele, "Treemaps with bounded aspect ratio," *Computational Geometry*, vol. 47, no. 6, pp. 683–693, 2014.

[37] J. Görtler, C. Schulz, D. Weiskopf, and O. Deussen, "Bubble treemaps for uncertainty visualization," *IEEE TVCG*, 2017.

[38] S. Hahn, J. Bethge, and J. Döllner, "Relative direction change - a topology-based metric for layout stability in treemaps," in *International Conference on Information Visualization Theory and Applications*, 01 2017, pp. 88–95.

[39] M. Sensalire, P. Ogao, and A. Telea, "Evaluation of software visualization tools: Lessons learned," in *Proc. IEEE VISSOFT*, 2009.

[40] A. Seriai, O. Benomar, B. Cerat, and H. Sahraoui, "Validation of software visualization tools: A systematic mapping study," in *Proc. IEEE VISSOFT*, 2014.

[41] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz, "A systematic literature review of software visualization evaluation," *J Syst Softw*, vol. 144, pp. 165–180, 2018.

[42] P. Steinhardt, "*libgit2* API for Git repository management," 2018, https://libgit2.github.com.

[43] SciTools, "Understand static code analysis tool," 2017, https://scitools.com.

[44] R. da Silva, E. Vernier, P. Rauber, J. Comba, R. Minghim, and A. Telea, "Metric evolution maps: Multidimensional attribute-driven exploration of software repositories," in *Proc. Vision, Modeling, and Visualization (VMV)*. Eurographics, 2016, pp. 54–62.

[45] E. Parzen, "On estimation of a probability density function and mode," *Annals of Mathematical Statistics*, vol. 33, no. 3, pp. 1065–1087, 1962.

[46] G. van Rossum, "The Python programming language," 2017, https://github.com/python/cpython.

[47] The Authors, "Dynamic treemap for software evolution visualization benchmark," 2017, https://github.com/vissoft18/treemaps.

[48] S. Hachul and M. Jünger, "An experimental comparison of fast algorithms for drawing general large graphs," in *Proceedings of the 13th International Conference on Graph Drawing*, ser. GD'05. Springer-Verlag, 2006, pp. 235–250.

[49] G. D. Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu, "An experimental comparison of four graph drawing algorithms," *Computational Geometry*, vol. 7, no. 5, pp. 303–325, 1997.