A Stable Greedy Insertion Treemap Algorithm for Software Evolution Visualization

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

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

***Abstract*—Computing treemap layouts for time-dependent (dynamic) trees is an open problem in information visualization. In particular, the constraints of spatial quality (cell aspect ratio) and stability (small treemap changes mandated by given tree-data changes) are hard to satisfy simultaneously. Most existing treemap methods focus on spatial quality, but are not inherently designed to address stability. We propose here a new treemapping method that aims to jointly optimize both these constraints. Our method is simple to implement, generic (handles any types of dynamic hierarchies), and fast. We compare our method with 14 state of the art treemaping algorithms using four quality metrics, over 28 dynamic hierarchies extracted from evolving software codebases. The comparison shows that our proposal jointly optimizes spatial quality and stability better than existing methods.**

I. Introduction

Understanding the evolution of large and long-lasting software projects is a major aspect of program comprehension. Typically, evolution data for such projects is mined by fact extraction tools from existing software control management systems handing software repositories, such as Git [1], Subversion [2], and CVS [3]. Several types of data attributes are collected (and explored) in this way, including the identity of software items of interest (*e.g.*, packages, folders, files, classes, methods), various quality attributes measured on them (*e.g.*, testability, maintainability, modularity, and readability metrics [4]), and relations that interrelate these items. *Hierarchy* relations, which describe the containment or aggregation of software items, play a central role in virtually all such evolution analyses, since they offer a powerful and natural way to examine the (typically large) evolution data at multiple levels of detail. As such, methods that can depict time-dependent hierarchies are a central element of the program evolution toolset.

Dynamic, or time-dependent, treemaps are one of the most effective techniques for displaying time-dependent hierarchies. Compared to other techniques, such as node-link tree layouts, they use basically every pixel of the available screen space to display information, and as such scale to tens of thousands of items (tree nodes) per time step. Many treemap methods exist for handling static (time-independent) hierarchies [5], [6], which also have been shown to optimize various quality measures that help readability, such as aspect ratio [6], [7] and relative positions of nodes [8]–[12]. However, far fewer methods are available for dynamic trees [13]–[15]. One key problem for dynamic treemapping is *instability*, *i.e.*, the fact that relatively small changes in a tree can induce disproportionately large changes in the resulting treemaps. Finding a good way to quantify and reduce instability is an open problem for dynamic treemap algorithms.

In this paper, we address the above limitations with two main contributions. Firstly, we propose a new dynamic treemap algorithm, called Greedy Insertion Treemap (GIT). GIT aims to preserve treemap-cell neighborhoods over time by constructing an initial so-called Layout Tree (LT), which is next incrementally updated as the tree data changes, so as to minimize undesired treemap-layout changes. Secondly, we evaluate the quality of GIT both in the spatial domain and the temporal domain against a large set of well-known treemap algorithms using several established quality metrics, and on a large set of dynamic hierarchies extracted from real-world software repositories. Our evaluation results show that GIT strikes a better balance between spatial and temporal quality than the existing competing methods we evaluated against. As GIT has a simple and computationally scalable implementation, we argue that it represents a valuable contribution to the toolset of techniques needed by program evolution comprehension.

The structure of this paper is as follows. Section II outlines existing work on (dynamic) treemapping and related quality metrics, and their use in program evolution comprehension, and also introduces the treemap methods we compare against. Section III details our new GIT algorithm. Section IV presents our evaluation methodology for GIT and the obtained results are revealed in Section V . Section VI discusses our proposal and outlines directions for future improvement.

II. Related Work

In this section, we will discuss the Algorithms (Section II-A) and Quality Metrics (Section II-B) present in the dynamic treemap literature. Let $T = \{n_i\}$ be a hierarchy (tree) with nodes n_i, each having a weight value $a_i \geq 0$. Weights are given for leaf nodes and computed for non-leaf nodes as the sum of their children weights, respectively. Let $\mathcal{T}(T)$ be the treemap layout of T, with a rectangle cell r_i assigned to each n_i, so that the area of r_i equals a_i.

A. Algorithms

Time-dependent hierarchies $T(t)$ are a central artifact to explore in program evolution comprehension. Since such analyses usually involve tens or even hundreds of time steps t, small-multiple visualizations (one image per time step) do not scale well, hence showing an animated layout of the changing hierarchy is preferred [16]. For this, several techniques construct a so-called union tree $\cup_t T(t)$, build a single layout of this union tree, display it using Icicle plots [17] or Sunburst diagrams [18], and then highlight changes of $T(t)$ over time in it [19]. While this approach minimizes instability (layout changes) over time, and is simple to implement, it cannot handle long time sequences and/or large trees.

Treemaps cope well with the need for handling large trees [20]–[23]. Slice and dice (SND) treemaps introduced the idea but were found to create poor aspect-ratio (AR) cells which are hard to see [5]. Squarified treemaps (SQR) propose a heuristic that yields good (close to one) AR values [6]. A subsequent algorithm (APP) was designed to approximate the optimal AR [7]. While treemaps were originally designed to handle time-independent trees, the need for *stability* was soon revealed – that is, small changes in the input tree T should yield only small changes in the treemap $\mathcal{T}(T)$. Several algorithms were designed to improve stability. Ordered treemaps (OT) [9] and Strip treemaps (STR) [24] lay out cells r_i using a given order of the nodes of T, using different heuristics – Pivot-by-Middle (PBM), Pivot-by-Size (PBZ), and Pivot-By-Split-Size (PBS) [9]. Other algorithms lay out cells along a space-filling fractal-like curve, *e.g.*, Spiral (SPI) [25], and Hilbert (HIL) and Moore (MOO) methods [26]. Yet another ordering technique considers node similarities: Spatially-Ordered Treemaps (SOT) [8] processes sibling nodes ordered by decreasing similarity; NMap [12] places cells according to the similarity of their nodes using dimensionality reduction. Variants thereof include NMap Alternate Cuts (NAC), which splits the screen space alternating horizontal and vertical slices; and NMap Equal Weights (NEW) which aims to create similar-size cells.

Stability becomes a major concern when treemapping time-independent trees with potentially long evolution and large variations. However, only a few methods explicitly aim to treat dynamic data. Stable treemaps [14] aim to improve both AR and stability by using non-sliceable layouts. However, this method is computationally expensive and not trivial to implement. Voronoi treemaps [27], [28] achieve, in general, good AR values, and have been adapted to also handle dynamic trees to visualize software structure evolution [29], [30]. There exist also methods that propose other cell shapes, or combinations of multiple shapes, such as bubble treemaps [31], jigsaw treemaps [32], and orthoconvex treemaps [33]. However, such methods have not been specifically designed with the aim of maximizing stability.

B. Metrics

As outlined in Sec. II-A, treemap quality consists of two main components:

Spatial quality captures how readable the treemap geometry is. The best known, and most used, metric for this is the aspect ratio (AR) of the cells r_i which should ideally reach one. The so-called readability metric measures how often a user's gaze changes direction while reading an ordered treemap along the predefined node ordering [24]. The continuity metric measures how often cells of nodes which are close in the given node ordering are far apart in the treemap [25].

Stability metrics capture how easily can a user understand the changing geometry of a dynamic treemap. This is measured essentially by quantifying the visual change $\delta(r_i(t), r_i(t+1))$ of the cells r_i, and then aggregating such visual changes into a single value using some function S. Early on, Shneiderman and Wattenberg [9] defined the Layout Distance Change metric, where they used for δ the distance between the vectors $(x_i(t), y_i(t), w_i(t), h_i(t))$ and $(x_i(t+1), y_i(t+1), w_i(t+1), h_i(t+1))$, x and y being the coordinates of the top-left corner, and w and h, the width, and the height of a rectangle r_i. They defined S as the average of δ for all cells and revisions. Later, Hahn *et al.* [13] use for δ the distance between the centers of $r_i(t)$ and $r_i(t+1)$ and also average for S. Tak and Cockburn [26] use for S the variance and define δ as [9]. They also propose a drift metric, which measures how much a cell's center moves away from its average position over long time intervals. Recently, we have seen new metrics that measure stability not by looking only at a single cell's position relative to its past states, but take into consideration the relationships between all cells in the layout. Hahn et al. [34] propose the relative direction change, which measures angle differences between all centroids in the layout between consecutive time-steps, and Sondag *et al.* [14] propose similar metric, where δ measures how a cell moves with respect to all its neighbors, where S is again the average. We will discuss these metrics further, and also propose a new one in Section IV-A.

III. Greedy Insertion Treemap

As outlined in Sec. II, many treemapping methods exist in the literature, and these have been evaluated by several metrics for both spatial quality and stability. However, examining the above in more detail, we find two limitations: (a) most existing treemap methods have been designed without the *explicit* aim of maximizing stability; (b) among the few methods where stability was an aim, there is no clear optimal method which yields both good spatial quality and stability for long time sequences of trees exhibiting a high dynamics in terms of node additions, deletions, and weight changes. We next propose a method, Greedy Insertion Treemap (GIT), that aims to outperform the current state-of-the-art in these two respects.

GIT is designed from the start with the aim of increased stability. For this, GIT aims to preserve cell neighborhoods in the treemap over time. To this end, we use a so-called Layout Tree (LT) help data structure (not to be confused with the tree T we want to visualize). Each node $l \in LT$ represents a treemap cell, and may have two subtrees: (a) $R(l)$ is rooted at the top-right corner of l; and (b) $B(l)$ is rooted at the bottom-left corner of l. Together with the cell weights, LT

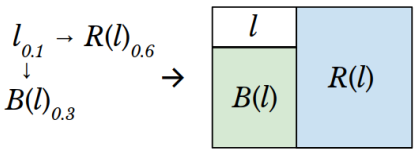

Fig. 1. Space partitioning from LT.

fully encodes a treemap $\mathcal{T}$. Indeed, we can construct $\mathcal{T}$ by traversing LT breadth-first. During this, for each $l \in LT$, we compute the total weight of its subtrees $R(l)$ and $B(l)$, and cut the remaining drawing space vertically and horizontally according to these summed weights, as illustrated in Fig. 1.

GIT proceeds in two phases: initialization and update, as follows.

Initialization: To start with, we need to construct LT from the first tree $T(t = 0)$ in our sequence. For this, we can use basically any method $\mathcal{T}_{init}$ that constructs a (static) treemap for $T(0)$ from which we can next generate LT with the properties (a) and (b) mentioned above. We have experimented with two such initialization methods. First, we constructed LT from a squarified treemap (*i.e.* $\mathcal{T}_{init}$ is the SQR algorithm), since SQR is well known to yield very good AR values. Alternatively, we propose a simple heuristic $\mathcal{T}_{init}^{direct}$ that directly builds LT from the initial tree $T(0)$. Both initialization methods are compared next in Sec. V.

To explain our heuristic $\mathcal{T}_{init}^{direct}$, consider a single-level tree $T = [(n_i, a_i)] = [(A, 10), (B, 2), (C, 8), (D, 4), (E, 1), (F, 3)]$, to be laid our, for simplicity, in a square drawing area S of size 1; handling general trees is trivial by top-down recursion. We build LT by sequentially adding each $n_i \in T$ to it (Fig. 2). After each addition, we rebuild $\mathcal{T}$ from the current LT as explained above, so it covers the entire S. Thus, existing nodes are 'squeezed' to make space for the new nodes. In our example, we first add node A, which will cover the entire S. To add B, we find the node $n \in LT$ having the worst aspect-ratio cell $c \in \mathcal{T}$. If $w_c \geq h_c$, we add B directly right of n (as in our example), else we add B directly below n, and update $\mathcal{T}$ from the new LT again. For the third node C, as the worst-aspect-ratio cell is B, and since $h_B > w_B$, we add C below B and update $\mathcal{T}$ from LT again. Fig. 2(d-f) shows the addition of the remaining nodes of T.

For didactic purposes, nodes were added in alphabetical order, but in reality, we want nodes to be added in random order, hence when dealing with truly hierarchical data, one sub-tree is not completely laid out before its siblings, which could cause it to be 'squeezed'.

Update: We now have an initial treemap $\mathcal{T}$ and its LT. We next edit LT to handle weight changes, node additions, and node deletions as T changes. *Weight* changes do not change LT. *Additions* are handled just as adding regular nodes when building the initial LT using $\mathcal{T}_{init}^{direct}$. Additions tend to increase the cells' aspect ratios, so we do them after node

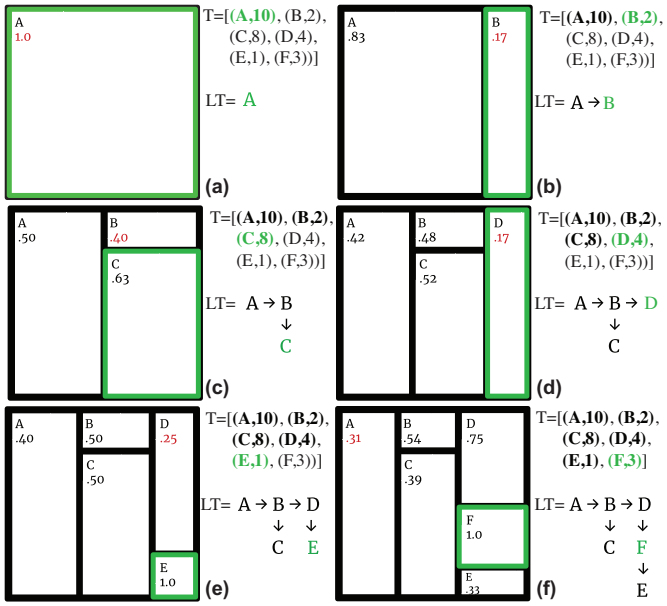

Fig. 2. Building the initial layout tree LT (green: inserted cells).

removals and weight changes. *Removals* are done by editing LT as follows (see also Fig. 3):

1) If a node $n \in LT$ has a $B(n)$ subtree, we replace n by $B(n)$;
2) else if n has a $R(n)$ subtree, we replace n by $R(n)$;
3) else n has no subtrees, so we just remove it from LT.

After handling all changes in a new revision of T, we rebuild $\mathcal{T}$ from LT, as already explained. As we show next in Sec. V, GIT scores a very good balance of spatial quality *vs* stability.

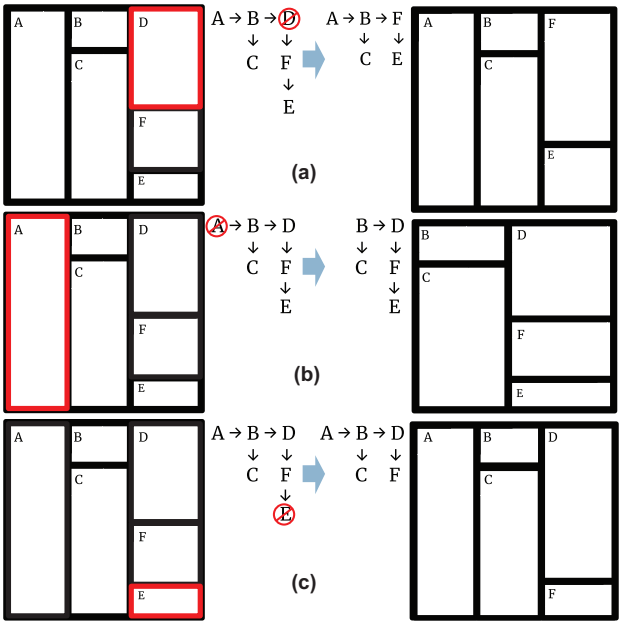

Fig. 3. Removal of nodes (red) from layout tree and its treemap.

IV. Evaluation

To evaluate GIT, we considered the following aspects:

A. Metrics

To evaluate the quality of GIT, we proceed as follows. For spatial quality, we use the well-known Aspect Ratio metric (AR) [6]. For each cell c in a treemap $\mathcal{T}$, $AR_c = min(w_c, h_c)/max(w_c, h_c)$. This metric is considered by virtually all rectangular treemap evaluations we are aware of.

For stability, we compute three metrics. The first two are the Shneiderman-Wattenberg's Layout Distance Change (LDC) [9] and Tak-Cockburn's Location Drift (LD) [26], already introduced in Sec. II. The LDC metric captures the instability of a cell between consecutive revisions. In contrast, the LD metric captures the deviation of a cell's position over all timesteps.

We also propose a (new) third metric, which extends LDC to also consider the change of the *data*. We define the *visual change* of a cell c_i as the Euclidean distance traveled by the four corners of the rectangle r_i between t and $t+1$, normalized by the treemap diagonal $\sqrt{W^2 + H^2}$, so $\delta v_i \in [0, 1]$. Next, we define the *data change* of c_i as $\delta a_i = |a_i(t) - a_i(t+1)|$, where a_i is the relative weight of node c_i. With these, we define the stability Q_i of a cell c_i in a treemap as

$$Q_i = (1 - \delta v_i)/(1 - \delta a_i). \quad (1)$$

We define the stability Q of an entire treemap as the average of its cells' stabilities Q_i. In contrast to LDC, Q measures how much a rectangle changes *in relation to its data change*. Measuring only absolute changes of rectangles (LDC) does not, we believe, fully characterize stability. Indeed, a rectangle could (and should) change a lot if its underlying cell's weight changes a lot. However, this does not mean necessarily that the treemap algorithm is unstable.

B. Techniques

We tested GIT against 14 other treemapping algorithms: Approximate (APP), Hilbert (HIL), Stable treemaps (LM0, LM4), 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). For NMap, we use as seed layout the one computed by SQR [12]. We did not consider non-rectangular treemap methods in the evaluation, since not all the metrics in Sec. II-B directly generalize to non-rectangular cells.

C. Datasets

We extracted 28 dynamic hierarchies by mining the structure of software projects (folders, files, classes) from 28 corresponding public GitHub repositories, using a custom automated pipeline that scans all available revisions and extracts the code structure using Understand [35]. As weights w_i, we use the number of lines of code of the respective items. Other software quality metrics delivered by Understand can be used instead, if desired. For more details on this process, we refer to [36]. The considered repositories have quite different sizes, number of revisions, hierarchy depths and shapes, number of developers, and code type (programming languages and application types). Statistics about the datasets are available in Table I.

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.

V. Results

We evaluate GIT on the aforementioned datasets, algorithms, and metrics collection from several perspectives, by answering a series of questions. Below, average stability S refers to the average of the LDC and Q metrics introduced in Sec. II-B. All results that we were not able to fit in the paper can be found at our online repository [37].

A. How does GIT's initialization affect its quality?

As outlined in Sec. III, we can initialize GIT with various treemap layouts, such as squarified (SQR) or using the direct initialization illustrated in Fig. 2. Intuitively, one would think that SQR initialization is to be preferred, since SQR is well known for its high AR values. To test this, we ran GIT using both initializations for all datasets. After initialization, the same regular GIT update mechanism is used in both cases. Figure 4 shows the per-dataset average stability and AR values. Interestingly, we see that the higher-AR SQR initialization actually yields slightly worse AR values for the entire sequence. For stability, the two initializations behave basically identically. We can explain this result by the fact that the GIT direct initialization follows the same heuristics as the update steps, while SQR forces GIT to start with a

layout which needs more substantial updates next as the tree data changes. At a higher level, this experiment suggests that GIT performs very well using direct initialization. As such, we use this initialization in all subsequent experiments.

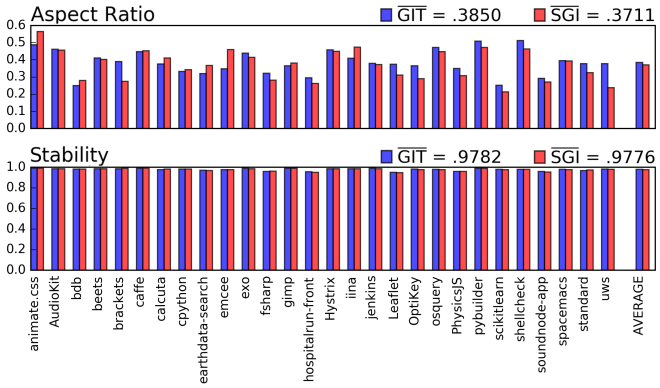

Fig. 4. GIT performance using $\mathcal{T}_{init}^{direct}$ (GIT) *vs* squarified initialization (SGI).

B. How do visual quality and stability vary over time?

As we have already noted, spatial quality and stability are roughly inversely correlated desiderates – a treemap that scores well for one of these metrics tends to score less well for the other one. Hence, comparing how these metrics change in time is interesting. To answer this, we display, for one dataset and all tested algorithms, two charts showing the median (black), 25-75% range (green), and 5-95% range (gray) of the AR and S metrics (Fig. 5). We see that APP and SQR have the best AR values, and SND the worst AR values. The other algorithms, including GIT, score in-between. In contrast, GIT, LM0, and SND score the best for stability, while all other algorithms exhibit a non-negligible number of unstable time moments. This suggests that GIT strikes a good compromise between stability and aspect ratio.

While Fig. 5(right) shows how the per-timestep stability changes over time, it does not show us which actual instability patterns each method is prone to deliver. Knowing this is useful, as we can better understand what to expect in terms of (undesired) cell moves from a certain algorithm, including GIT. To show this, we plot the trails connecting all centers $k_i(t)$ of all rectangles $r_i(t)$ for consecutive t values over a given tree sequence (Fig. 6). We set the opacity of each line segment $(r_i(t), r_i(t+1))$ to the Euclidean distance $\|k_i(t) - k_i(t+1)\|$ normalized by the square root of the number of time steps. Hence, dark long lines show big moves (instability) while small moves (close to stability) are hardly visible. The image confirms the high stability of GIT – in contrast to most other methods, except SND, GIT creates smaller cell moves (shorter dark lines), and most of these are close to horizontal or vertical. Interestingly, we see that other methods create quite different move patterns: SQR, PBS, PBZ, and PBM have mostly (large) diagonal moves. SPI shows a coil-like movement and in STR we see no vertical travel. Overall, we see that GIT is more stable not only because it yields smaller moves, but also because it constrains these to fewer motion directions, thus causes less complex dynamics (that the user must follow) in the resulting visualization. This can be also checked by watching the actual videos showing the algorithms in action [37].

C. How do all quality metrics vary over all datasets?

The experiments so far do not show the individual stability metrics (including LD, which can be only computed for an entire sequence), nor, for space reasons, the metrics over all 28 tested tree sequences. To get more insight in how GIT performs in these respects, we show the per-dataset average values (for AR and the three stability metrics) for all tested methods, all datasets (Fig. 7). Cells are colored using a purple (low values) to yellow (high values) colormap. We observe the following: For AR, APP scores consistently better for most datasets than all other tested methods. SQR reaches the highest AR values, but only for a very few datasets. SND, as expected, scores overall the poorest. The remaining methods can be divided roughly into two groups, with NEW, PBM, PBS, STR, and PBZ scoring overall higher than GIT, HIL, LM0, LM4, MOO, and NAC. Concerning stability, SND scores consistently the best for all three considered metrics, and GIT, LM0, and LM4 come in the second place. This strengthens our earlier observation that GIT strikes a good balance between stability and spatial quality.

D. How to summarize GIT's quality?

As noted, GIT seems to strike a good balance between spatial quality and stability. We summarize both these metrics for GIT and all other algorithms using a star plot (Fig. 8). The figure shows a scatterplot with x mapping average stability S and y aspect ratio AR, respectively. Categorically colored points, one color per method, indicate the tested methods, attributed by their S and AR values over all datasets, all time steps. From each point (method), we draw lines connecting it with the S and AR values obtained for all the 28 tested datasets. A good algorithm has thus its 'star' center placed top-right and relatively short star arms, indicating consistent quality over the entire dataset collection. We see several patterns, as follows.

At a high level, stability is roughly inversely correlated with spatial quality – methods that score very well on one tend to score worse on the other. We see three groups of methods: APP, PBS, SQR, PBM, STR, PBZ and NEW score well on spatial quality, but poorly (except NEW) on stability. SND is the opposite outlier, scoring best on stability but clearly poorest on spatial quality. A middle group of methods (GIT, LM0, LM4, MOO, NAC, SPI, and HIL) trades well stability *vs* spatial quality. Within these, GIT scores the best stability, and LM0 the best spatial quality. As such, GIT and LM0 can be considered complementary methods with respect to the stability *vs* spatial quality trade-off. However, LM0 has a considerably more complex and slower implementation than GIT – for details, we refer to [14]. Separately, we see that GIT's star size (convex hull containing the lines emerging from

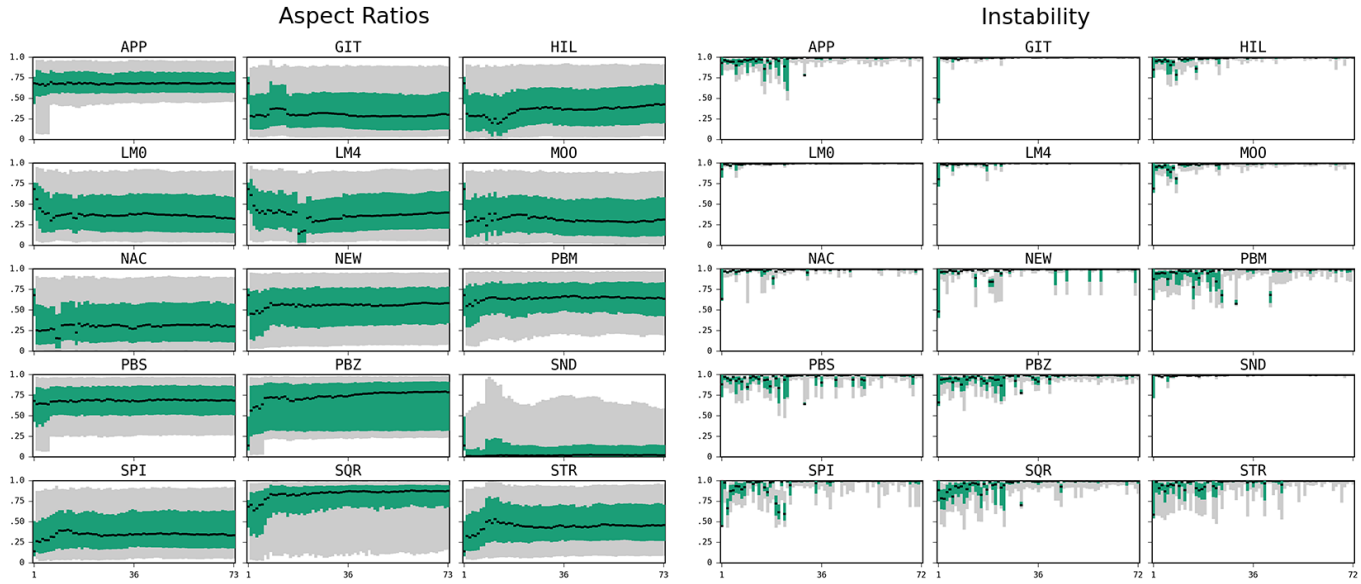

Fig. 5. Distribution of aspect ratio ($AR, left$) and average stability (S, right) values over time for the GIMP dataset.

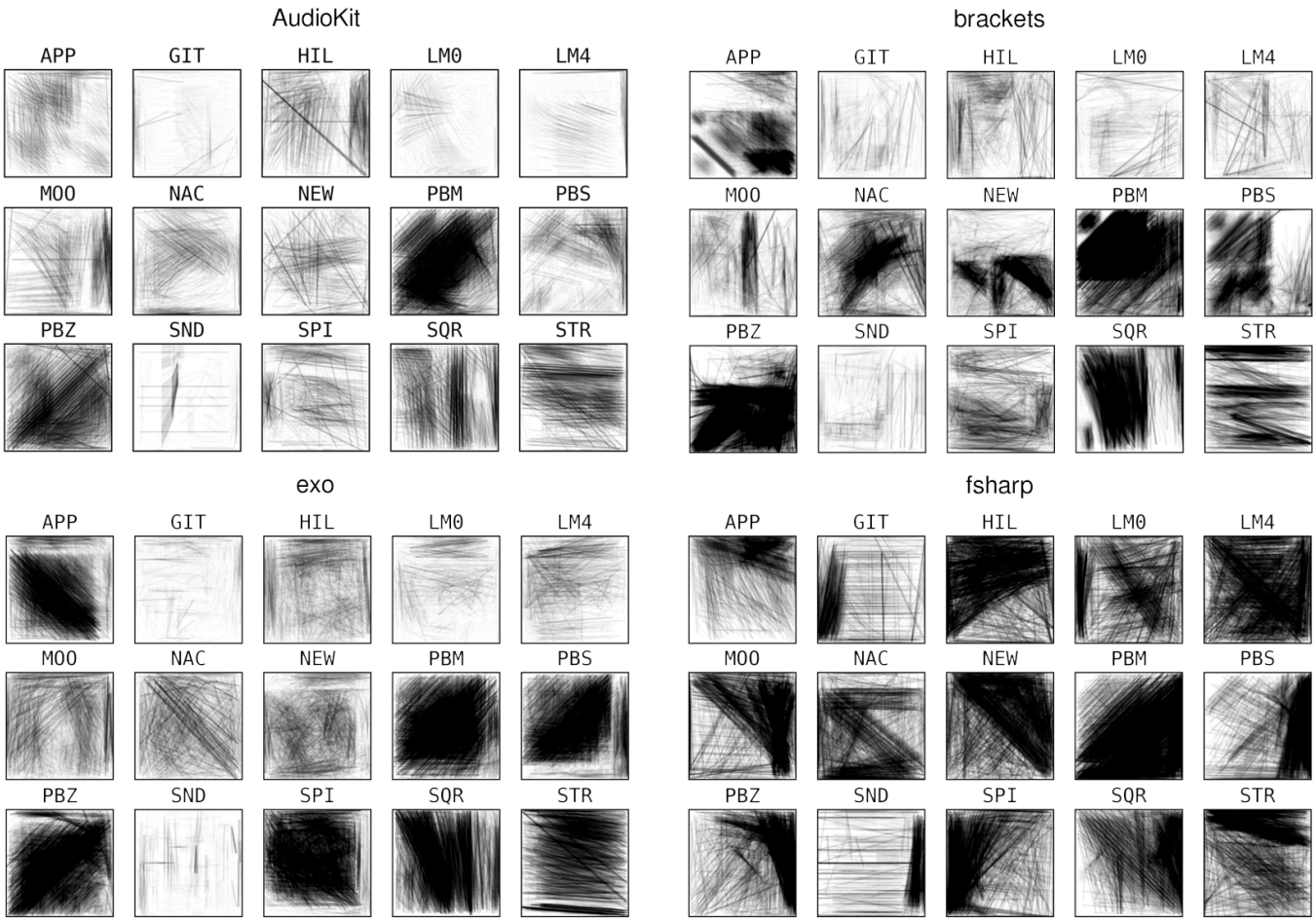

Fig. 6. Instability (cell center motion) patterns, all methods, *AudioKit*, *brackets*, *exo* and *fsharp* datasets.

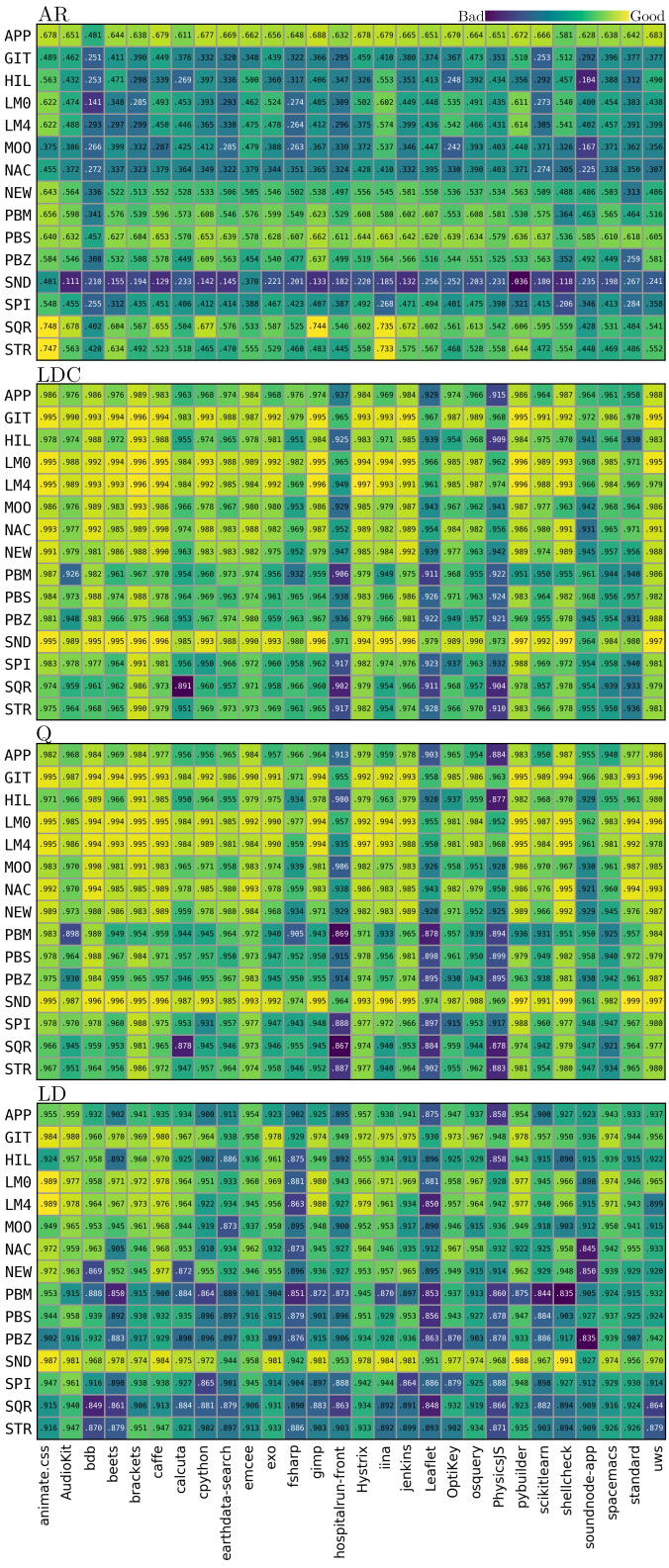

Fig. 7. Average metric values for all techniques and all datasets.

the GIT point) is one of the smallest of all tested methods. Hence, GIT offers one of the most consistent behaviors over the entire dataset collection from all tested algorithms.

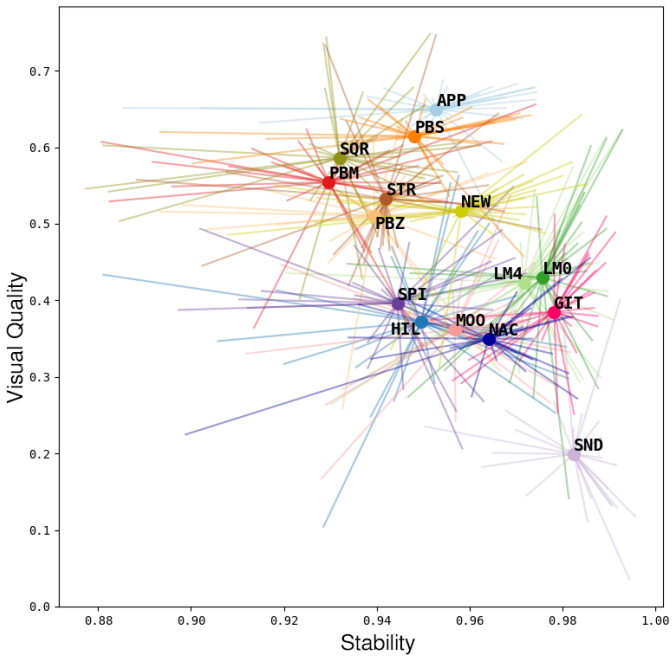

Fig. 8. Star plot summarizing both visual quality and stability, all algorithms, all datasets.

VI. Conclusion

We have presented a new method for computing treemap layouts for time-dependent hierarchies. As discussed earlier, there are only a few methods in the literature that consider quality aspects pertaining to *both* spatial quality and stability of such treemaps. Our contribution, in brief, is proposing a new method that takes both these quality aspects into account; and evaluating our method comprehensively on a broad dataset of 28 time-dependent hierarchies extracted from real-world dynamic dataset (software repositories), against 14 well-known treemapping methods, and using 4 quality metrics. Our results show that our new method strikes a good balance between spatial quality and stability as compared to state-of-the-art methods. Additionally, our method is simple to implement, fast, generic (with respect to the considered dynamic hierarchies), and has no hidden free parameters. More importantly, our method is an addition of a very small set of so-called *stateful* methods that consider the evolution of a dynamic tree sequence when computing suitable treemaps thereof. Most existing treemapping methods are not designed to consider tree state, which arguably makes them suboptimal for handling inherently stateful datasets like dynamic trees.

Several future work directions are possible, as follows. Firstly, it is interesting to extend our evaluation to dynamic hierarchies from other domains than software evolution. This may show how much our proposal can effectively handle such more diverse datasets. Secondly, we argue that more refined

quality metrics are needed (in general, for our new method but also any other treemapping methods) to capture the quality of such methods, as perceived by end users and in sync with their tasks. Finally, understanding the trade-off between the (algorithmic) reasons behind spatial quality and stability, *i.e.* what to do to optimally satisfy both these requirements, is an open problem, to which we believe to have contributed to with our current work.

REFERENCES

[1] "GIT source code management," 2018, https://git-scm.com/.

[2] "Apache Subversion – enterprise-class centralized version control for the masses," 2018, https://subversion.apache.org/.

[3] "CVS - Concurrent Versions System," 2018, https://www.nongnu.org/cvs/.

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

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

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

[7] 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.

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

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

[10] M. Ghoniem, M. Cornil, B. Broeksema, and M. Stefas, "Weighted Maps : treemap visualization of geolocated quantitative data," *Conference on Visualization and Data Analysis*, vol. 9397, no. February, pp. 1–15, 2015.

[11] K. Buchin, D. Eppstein, M. Löffler, M. Nöllenburg, and R. I. Silveira, "Adjacency-preserving spatial treemaps," in *Proceedings of the 12th International Conference on Algorithms and Data Structures*, ser. WADS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 159–170.

[12] 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.

[13] 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.

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

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

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

[17] J. B. Kruskal and J. M. Landwehr, "Icicle Plots: Better Displays for Hierarchical Clustering," *The American Statistician*, vol. 37, no. 2, pp. 162–168, may 1983.

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

[19] C. Hurter, O. Ersoy, and A. Telea, "Smooth bundling of large streaming and sequence graphs," in *2013 IEEE Pacific Visualization Symposium (PacificVis)*, Feb 2013, pp. 41–48.

[20] 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.

[21] B. Shneiderman and C. Plaisant, "Treemaps for space-constrained visualization of hierarchies," 2018, https://cs.umd.edu/hcil/treemap-history.

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

[23] 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.

[24] 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.

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

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

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

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

[29] 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.

[30] 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).

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

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

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

[34] S. Hahn, J. Bethge, and J. Dllner, "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.

[35] SciTools, "Understand static code analysis tool," 2018, https://scitools.com.

[36] 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.

[37] The Authors, "GIT – Dynamic Treemap Benchmark," 2018, https://github.com/sibgrapi18/treemaps.