# SizePairs: Achieving Stable and Balanced Temporal Treemaps using Hierarchical Size-based Pairing

Chang Han, Anyi Li, Jaemin Jo, Bongshin Lee, Oliver Deussen, Yunhai Wang

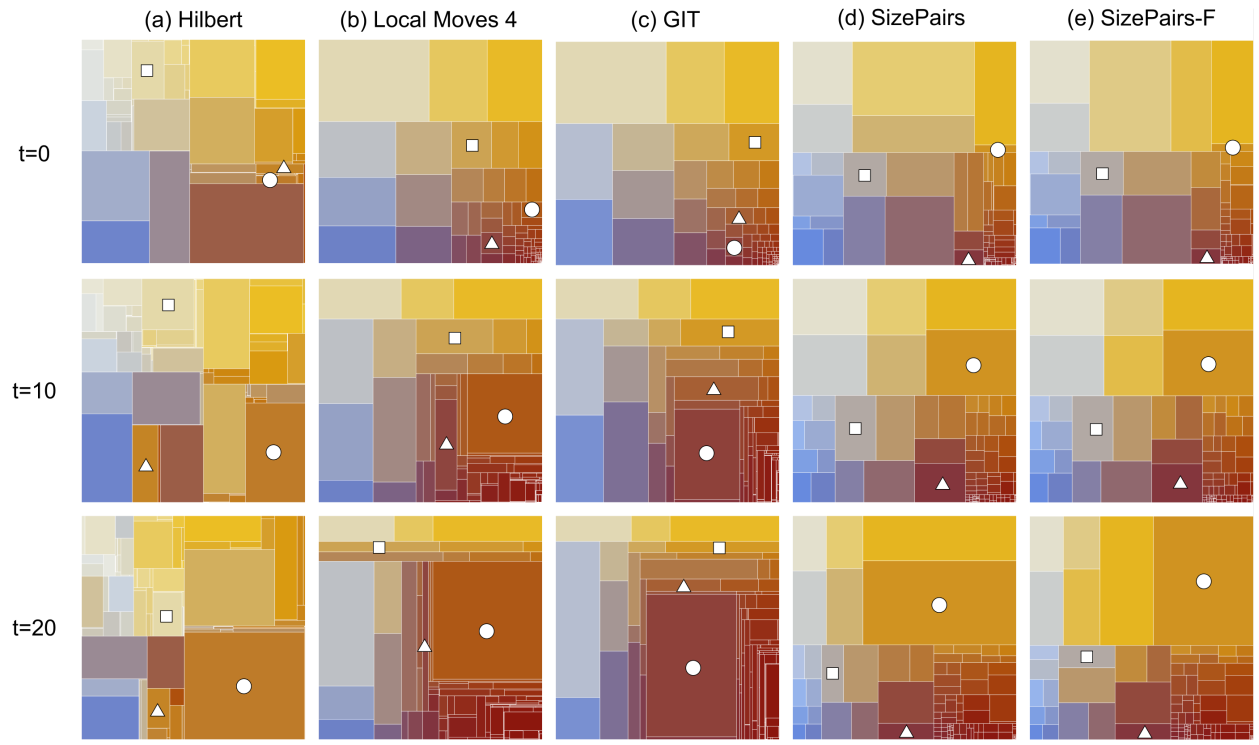


Fig. 1. Comparing different temporal treemap methods using three time steps (t = 0, t = 10, and t = 20) of the *WorldBankHIV* data: (a) Hilbert treemap method [26]; (b) Local moves method [25]; (c) Greedy insertion treemaps [29]; (d) SizePairs; and (e) SizePairs with local moves. To reflect the stability of each method, we assign colors to each rectangle (based on its position) at the first time step and then apply the color scheme to the other time steps (we mark three data items using shpaes to facilitate the comparison). SizePairs is more stable and maintains better aspect ratios, and SizePairs in combination with local moves (e) has even better aspect ratios than SizePairs alone (d).

**Abstract**—We present SizePairs, a new technique to create stable and balanced treemap layouts that visualize values changing over time in hierarchical data. To achieve an overall high-quality result across all time steps in terms of stability and aspect ratio, SizePairs employs a new hierarchical size-based pairing algorithm that recursively pairs two nodes that complement their size changes over time and have similar sizes. SizePairs maximizes the visual quality and stability by optimizing the splitting orientation of each internal node and flipping leaf nodes, if necessary. We also present a comprehensive comparison of SizePairs against the state-of-the-art treemaps developed for visualizing time-dependent data. SizePairs outperforms existing techniques in both visual quality and stability, while being faster than the local moves technique.

**Index Terms**—Treemaps, Stability, Compensation, Temporal Treemaps

## 1 Introduction

- *C. Han, A. Li, and Y. Wang are with Shandong University. E-mail: {hatch.on27, lianyisdu, cloudseawang}@gmail.com.*
- *J. Jo is with Sungkyunkwan University. E-mail: jmjo@skku.edu.*
- *B. Lee is with Microsoft Research. E-mail: bongshin@microsoft.com.*
- *O. Deussen is with University Konstanz. E-mail: oliver.deussen@uni-konstanz.de*
- *Y. Wang is the corresponding author.*

*Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx*

Treemaps [15] are a space-filling technique to visualize hierarchical data. They depict a hierarchy as nested rectangles, where each node in the hierarchy is shown as a rectangle, while the size of leaf nodes encodes a numerical value. Because of their efficient use of space, treemaps have been successfully used in many practical applications (e.g., [14,18,30]). In addition to visualizing a static hierarchy, treemaps can be used to visualize hierarchies that evolve over time (i.e., temporal treemaps) where the numerical value associated with a leaf node changes and nodes are inserted or deleted at a certain time point.

A simple way to construct temporal treemaps is to build a treemap separately for each time step by using any existing static treemap layout method and combine them. However, the treemaps resulting from most methods are difficult to compare visually because the same data item might appear in different locations at different time steps. Although the Slice-and-Dice method [23] can ensure stability, the visual quality of its resulting layout is poor. To take stability into account, Shneiderman and Wattenberg [24] presented ordered treemaps which ensure that items always are placed close to each other in the treemaps over time. Although such treemaps and their variants [1, 26, 27] greatly improve stability, preserving the data order cannot be guaranteed. When data changes over time, neighboring items in the order often are not placed closeby in a completely re-computed treemap (e.g., Fig. 1(a) by the Hilbert method [26]). Moreover, for complex data these methods often result in the treemaps with poor aspect ratios.

To address this issue, researchers proposed state-aware treemaps, such as Local Moves (LM) [25] and Greedy Insertion Treemaps (GIT) [29], which control stability by using the treemap of the previous time step to compute the next one. LM adapts the treemap via local modifications (i.e., flip and stretch operations), while GIT incrementally updates the treemap of the first time step, maintaining a tree structure. Both methods substantially improve the stability between adjacent time steps while maintaining high visual quality. However, since they rely on the treemap of the first time step, the visual quality and long-term stability of such treemaps decrease over time if data changes significantly (Figs. 1(b,c)), especially for frequent insertions and deletions.

In this paper, we introduce SizePairs, a new treemap layout algorithm to generate a series of high quality layouts, while maintaining short- and long-term stability. Instead of starting from the layout of the first time step, SizePairs first constructs a global layout tree by considering the entire time steps, and uses this tree to derive the layout of each time step. SizePairs employs a new hierarchical pairing algorithm that recursively pairs two nodes that complement their size changes over time and have similar sizes. For example, if one data item is growing and the other is shrinking, pairing them in a hierarchy limits the impact of the data change to a local region, thus making the global layout more stable. To avoid rectangles with poor aspect ratios, we further ensure paired items to have similar sizes over the whole time. Starting with each data item as an individual leaf, SizePairs iteratively pairs and joins nodes until one node is left. During this procedure, nodes with similar sizes are preferably paired together for achieving a good visual quality. Once a binary pairing tree is built, SizePairs aggregates the temporal changes of each node into one value and uses the aggregated results to select a splitting orientation (vertical or horizontal) for each internal node to achieve better aspect ratios. As a result, a global layout tree is constructed for single-level treemaps, while the one for multi-level treemaps can be constructed in a recursive way.

Based on the pre-computed global layout tree, SizePairs allows for an interactive generation of a stable and balanced treemap for each time step (see Fig. 1(d)). Furthermore, SizePairs includes an optional step that allows to apply local moves operations (i.e., flip) to the leaf nodes of the layout tree for further improving its visual quality, see the area marked by the white circle in Fig. 1(e). Although this step might reduce stability, within each internal node the stability is maintained and thus the resulting instability is typically not perceived. In addition, users are allowed to edit the global layout tree to interactively customize treemap layouts for satisfying application requirements in a stable way, such as item re-ordering and label placement.

We compared SizePairs with the state-of-the-art methods using a large set of real-world datasets as benchmarks. The results show that SizePairs performs the best in terms of the short- and long-term stability measures—corner-travel distance and normalized location drift—and the mean aspect ratio, regardless of data characteristics (e.g., the levels of hierarchy). Regarding runtime, it is similar to GIT but is nearly one order of magnitude faster than LM. Our qualitative evaluation also reveals that the visual quality of layouts that SizePairs generated is more consistent throughout all time steps.

In summary, our contributions are twofold: (1) SizePairs, a new layout method that generates stable and balanced temporal treemaps by introducing a size-pairing based global layout tree and (2) a comprehensive evaluation that shows SizePairs outperforms the state-of-the-art methods in terms of short- and long-term stability and visual quality.

## 2 Related Work

A few techniques have been developed for visualizing evolving hierarchies, which are mainly based on two visual representations: temporal treemaps and nested streamgraphs [2, 6, 17, 19]. The former generate a series of treemaps with good stability over time for showing changes in the data, while the latter attempt to convey the overview of the whole data in a single view based on the ThemeRiver metaphor [4, 12]. Here, we focus on temporal treemaps [22], specifically on quality metrics and methods for generating them.

### 2.1 Quality Metrics

Temporal treemaps are mainly evaluated in terms of two criteria—visual quality and stability—and a few corresponding metrics have been introduced.

**Visual Quality.** The visual quality of a treemap is usually defined as the mean aspect ratio of the rectangles in the treemap. As shown in Kong et al. [16], graphical perception for comparing rectangles with extreme aspect ratios (e.g., thin elongated rectangles with aspect ratio of $\frac{9}{2}$) is known to be inaccurate. Therefore, most of the methods have attempted to avoid extreme aspect ratios. Although extreme aspect ratios are definitely considered ineffective, the optimum aspect ratio for rectangles is still under investigation. Traditionally, an aspect ratio closer to 1 (i.e., a square) has been regarded better and has long served as an optimization objective for many methods. For example, Squarified treemaps [3] and Approximation treemaps [20]. Squarified treemaps use an effective heuristic to achieve near-optimal (i.e., close to 1) aspect ratios, and Approximation treemaps are mathematically proven to ensure a bound for the worst aspect ratio. Both methods produce treemaps with good aspect ratios in practice.

However, user studies [13, 16] revealed that using other aspect ratios, such as $\frac{3}{2}$ or $\frac{2}{3}$, can lead to more accurate comparison. In this paper, we do not optimize a layout towards a certain aspect ratio. However, we found the mean aspect ratio of layouts generated by our technique was the closest to $\frac{2}{3}$ compared to other state-of-the-art methods (Sect. 4).

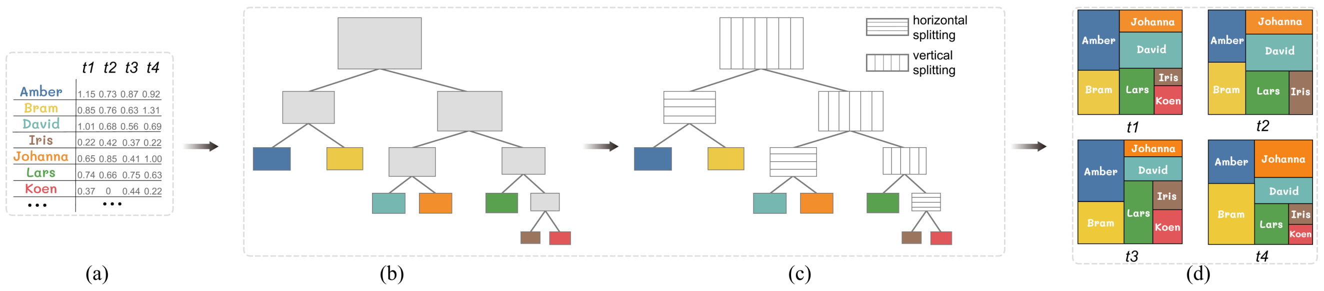
**Stability.** The stability of treemaps quantifies how much the same item in different layouts is consistently located. Various measures have been introduced to measure the stability between two layouts. Shneiderman and Wattenberg [24] used the Euclidean distance between the vectors $(x, y, w, h)$ of two rectangles, where $x$ and $y$ are the coordinate of the top left corner of each rectangle and $w$ and $h$ are the width and height, respectively. A simplified metric [9, 11] was also used, only measuring the distance between the centroids of the rectangles not considering their sizes. Another measure, popular in the computer vision field, is the corner-travel distance [28] that is defined as the sum of the distances between the four corners of two rectangles. These metrics are limited to capturing the stability between two adjacent layouts (i.e., short-term stability). To address this, Tak and Cockburn [26] proposed a location drift that measures how a rectangle is placed similarly throughout all layouts (i.e., long-term stability). In our paper, we use the corner-travel distance and normalized location drift to investigate both short- and long-term stability.

### 2.2 Temporal Treemaps

A trade-off exist between visual quality and stability when generating temporal treemaps. To srike the balance between them, various methods have been proposed; they can be grouped into two classes [28]: ordered treemaps and state-aware treemaps.

**Ordered Treemaps.** Assuming that there is an ordering for all data items, ordered treemaps [24] maintain a certain level of stability by placing items closer to each other in the input data nearby in the layout. A wide variety of ordered treemaps, such as [1, 7, 10, 11, 26, 27] have been introduced. However, we did not include them in our evaluation, since keeping the order of data does not necessarily produce stable layouts and the methods generally exhibited poor trade-offs between visual quality and stability [28].

**State-aware Treemaps.** The most relevant works to ours are state-aware treemaps [25, 29], recent methods that offer good trade-offs



| | t1 | t2 | t3 | t4 |
|---|---|---|---|---|
| Amber | 1.15 | 0.73 | 0.87 | 0.92 |
| Bram | 0.85 | 0.76 | 0.63 | 1.31 |
| David | 1.01 | 0.68 | 0.56 | 0.69 |
| Iris | 0.22 | 0.42 | 0.37 | 0.22 |
| Johanna | 0.65 | 0.85 | 0.41 | 1.00 |
| Lars | 0.74 | 0.66 | 0.75 | 0.63 |
| Koen | 0.37 | 0 | 0.44 | 0.22 |
| ... | ... | | | |

Fig. 2. A pipeline of our approach. (a) The input is a set of data items whose values vary at different time steps; (b) the items are organized by a binary tree, where the items with inverse trends and similar sizes are combined together; (c) computing the splitting orientation of each node in the binary tree, which is applied to all time steps for computing the layout; (d) the treemaps at four time steps.

between visual quality and stability. These methods reuse the layout for the previous time step to compute the current one, usually achieving better stability than indirectly keeping the data order. Sondag et al. [25] first introduced an incremental algorithm for state-awareness where an inital layout is generated by the Approximation treemap method and is locally modified, such as stretch and flip (i.e., local moves), to generate the next layout. The drawbacks of this incremental algorithm are two-fold. First, it generates an initial layout only using the data in the first time step but the first layout may not be suitable for the weights in other time steps; thus the visual quality often decreases over time. Second, it uses a hill-climbing [8] method to optimize the layout, but this often takes too long; we found out that it is about ten times slower than SizePairs.

Another state-aware algorithm is Greedy Insertion Treemap (GIT) by Vernier et al. [29]. It creates an initial layout using the Squarified treemap method and then constructs a binary tree (i.e., layout tree) of the data items based on the layout where a left (right) edge indicates a vertical (horizontal) split. Insertions and deletions are handled as tree operations on the layout tree. We found that GIT outperforms most previous methods in general although its algorithm is very straightforward. In Sect. 4, we show that SizePairs outperforms GIT in terms of visual quality, stability, and computation time.

## 3 SizePairs Treemaps

Our work aims to create stable and balanced treemaps for time-varying hierarchical data. We identify three design goals for our technique based on the design principles on treemap visualizations and the lessons from earlier work (i.e., LM [25] and GIT [29]).

- **DG1.** Create the layout of each time step as square as possible;
- **DG2.** Maintain the stability over time as much as possible; and
- **DG3.** Generate temporal treemaps as fast as possible.

The key challenge in achieving these design goals is that there are conflicts and trade-offs between them. For example, LM is optimized only partially for DG1 and DG2, where the visual quality and long-term stability decrease over time (see an example in Fig. 1(b)). Moreover, its computation cost is too expensive to make the method suit for interactive application. On the other hand, GIT is a solution for DG2 and DG3 by rapidly updating a layout tree computed from the first time step. However, because the layout tree may not fit to the other time steps as data changes, it is likely to fail to meet DG1 (see an example in Fig. 1(c)).

Instead, SizePairs is optimized both for DG1 and DG2 in a two-step procedure. First, it organizes all data items into a binary tree where all paired nodes are selected to complement their size changes over time and have similar sizes. In doing so, rectangles formed by the paired nodes have reasonable visual quality and good stability over time. Then, SizePairs strives to meet DG1 by choosing a better splitting orientation for each internal node and DG2 by using the global layout tree to compute the layout for each time step. Regarding the computational cost (DG3), SizePairs is lower than LM and comparable to GIT, especially for multi-level hierarchies (see Section 4). Since SizePairs needs the entire data to construct the binary tree, it can only deal with offline data.

Fig. 2 shows the computation pipeline of SizePairs. Taking a normalized $n \times m$ weight matrix $\mathbf{W}$ with a hierarchy on $n$ data items and $m$ time steps (Fig. 2(a)) as the input, SizePairs first constructs a binary tree for all items (Fig. 2(b)), followed by computing the splitting orientation of each internal node to form the global layout tree (Fig. 2(c)). After that, it uses this tree to compute the layout for each time step (Fig. 2(d)). SizePairs includes an optional step, applying local moves to further improve the visual quality of each time step. We refer to these two variants as SizePairs and SizePairs-F, respectively. Figs. 1(d,e) show the results generated by these two variants. Once the treemaps are generated, users are allowed to interactively edit the global layout tree to re-generate the treemaps that meet application-specific requirements.

### 3.1 Hierarchical Size-based Pairing

Given a set of data items, we construct the global layout tree by hierarchically merging pairs of items if they mutually compensate for their changes over time while having similar sizes. Fig. 3 illustrates this process. In Fig. 3(a) the weight of “Dave” increases and that of “Carol” decreases and this way would be a potential compensating partner, while the item “Alice” disappears and the weight of “Bob” increases to complement the reduced space. In addition to such compensation, we also encourage the items with similar sizes to be merged, which prevents one rectangle from being squeezed too much due to the change in the other. For example, the trend of “Carol” is better compensated by the that of “Eve” but their size difference are too large to produce rectangles with good aspect ratios (Fig. 3(b) vs. Fig. 3(d)). Accordingly, we choose the nodes to be merged in terms of two criteria: i) their *compensation degree* is larger than the other node pairs, and ii) their *size difference* is not large so as to maintain the visual quality. Before starting the pairing process, we take all data items as the leaf nodes of the global layout tree.

**Compensation Degree.** Given two nodes $i$ and $j$, we define the compensation degree based on the change of each item between consecutive time steps:

$$CD(i,j) = \frac{1}{m-1}\sum_{t=1}^{m-1} \frac{|(\mathbf{W}_{i,t+1} - \mathbf{W}_{i,t}) + (\mathbf{W}_{j,t+1} - \mathbf{W}_{j,t})|}{\max(\mathbf{W}_{i,t}, \mathbf{W}_{i,t+1}) + \max(\mathbf{W}_{j,t}, \mathbf{W}_{j,t+1})} \quad (1)$$

which is a value between 0 and 1. When $CD(i,j)$ is zero, these two items have completely inverse trends and thus compensate with each other (see Figs. 3(b,c)). Namely, the rectangle corresponding to the merged node remains stable over time. In contrast, pairing these two nodes will cause instability when $CD(i,j)$ is close to 1.

**Size Difference.** Given two nodes $i$ and $j$, we compute their size difference by:

$$SD(i,j) = \frac{1}{m}\sum_{t=1}^{m} \frac{|\mathbf{W}_{i,t} - \mathbf{W}_{j,t}|}{\max(\mathbf{W}_{i,t}, \mathbf{W}_{j,t})} \quad (2)$$

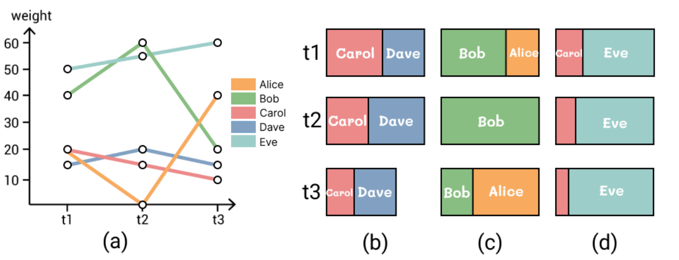


Fig. 3. Temporal compensating effect: (a) line chart showing the values of five data items over time; (b-d) show different pairing results: (b) pairing rectangles "Carol" and "Dave" which vary over time; (c) pairing "Alice" and "Bob"; and (d) pairing "Carol" and "Eve", which results in poor aspect ratios for Carol.

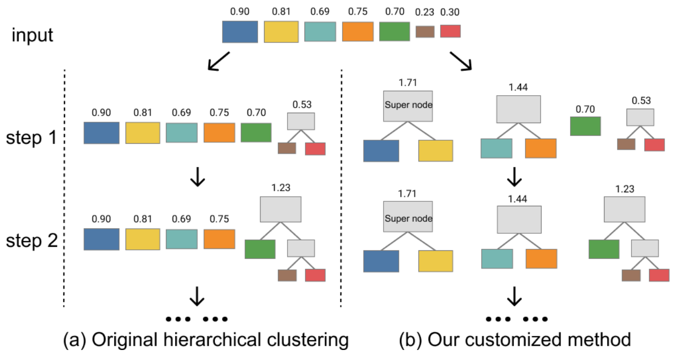


Fig. 4. Comparing two hierarchial pairing procedures with the data shown in Fig. 2. (a) Traditionally, only a single node is merged at each iteration; (b) our customized process pairs all non-super nodes at each iteration.

which ranges in [0,1]. A small $SD(i,j)$ means that two nodes have similar sizes, indicating that both small rectangles within the merged rectangle have good aspect ratios (see Fig. 3(b)). In contrast, a large $SD(i,j)$ means that packing two nodes together is likely to form the rectangle with extreme aspect ratios, see Fig. 3(d).

Based on these two terms, we define the cost between two nodes as:

$$C(i,j) = \omega\, CD(i,j) + (1-\omega)\, SD(i,j) \tag{3}$$

where $\omega$ is a constant. In our experiment, we found that $\omega = 0.5$ works well.

**Hierarchical Pairing.** Although the second term in Eq. 3 is helpful in selecting similar sized nodes to pair, it cannot prevent the case that two nodes with the smallest total cost are selected. To reject extreme cases, we require that the nodes with the sizes being larger than one third of the whole area can only be merged to the ones with similar sizes. For short, we refer such nodes as super nodes. This is inspired by the Approximation treemaps [20], which recursively split the rectangle into two sub-rectangles with the area ratio close to 1:2. To fulfill this goal, we propose a customized hierarchical clustering algorithm.

After initializing each item as a leaf node, we compute the cost between all node pairs and calculate the area of each node as the median of the corresponding time series. Next, we select super nodes based on their areas and put them into another list if there are. Then, we choose the pair from the remaining non-super nodes with the smallest cost to merge as a new node. We repeat this process until there is only one non-super node. Finally, super nodes and the left node (if there is one), are merged in the same way. Fig. 4(a) illustrates the first two steps of applying the hierarchical pairing method to the data shown in Fig. 2. The overall time complexity of this algorithm is $O(n^3 m)$, where $n$ is the number of data items and $m$ is the number of time steps.

To reduce the computation cost, we do not immediately compute the cost between the newly merged node and the other nodes after merging but update the cost matrix once all non-super nodes are paired

**Algorithm 1** Algorithm for hierarchical paring

```
Require: A set of data items $\{p_1, \cdots, p_n\}$ and a weight matrix $\mathbf{W}$
Ensure: The binary pairing tree $T$
 1: Initialize each leaf node with one data item
 2: Put all leaf nodes into an empty list $P$
 3: Initialize an empty list $L$
 4: Compute the median weight among all time steps for each node
 5: Compute the cost matrix for nodes in $P$
 6: repeat
 7:     Find the super-nodes in $P$ and put them into $L$
 8:     Merge the remaining nodes in $P$ in pairs
 9:     Update the cost matrix for nodes in P
10: until There is only one node $v$ in $P$
11: Put $v$ into $L$
12: Compute the cost matrix for all nodes in $L$
13: repeat
14:     Merge the two nodes with the smallest cost
15:     Update the cost matrix
16: until There is only one node
17: Take the only remaining node as the root of $T$
18: return $T$
```

or only one node is left (see Fig. 4(b)). In other words, the maximal number of computing operations is $n^2 m/4^i$ at the $i$th iteration. Doing so, a balanced hierarchy is created with the overall time complexity $O(n^2 m)$, while further alleviating the case that merging between large and small-sized nodes. In our experiment, we found that our algorithm is faster than LM and GIT when a hierarchy is present in the input data. For more detailed comparison, please refer to Sect. 4.

**Multi-level Treemaps.** For the input data with a hierarchy, we apply the above hierarchial pairing process from the root to the leaf level separately. Fig. 5 shows an example, where the pairing tree shown on the right preserves the topology of the input hierarchy. Since the pairing process is done among the nodes with the same parent, it even takes less cost than pairing single-level treemaps. For example, we only need to perform pairing twice among four nodes and once among three nodes in Fig. 5, while the single-level treemap in Fig. 2 takes more time to pair 7 nodes.

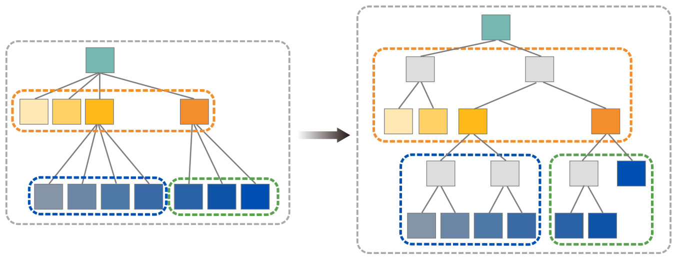


Fig. 5. (Left) An input hierarchy; (Right) A binary pairing tree computed from the input hierarchy.

## 3.2 Interactive Treemap Generation

Once the binary pairing tree $T$ is constructed, we need to convert it into a global layout tree by determining the splitting orientation for each node and then use this global tree to guide the treemap layout for each time step.

**Splitting Orientation.** Given a rectangle $R$, we recursively split it into two parts $R_l$ and $R_r$ corresponding to the two nodes in $T$. Since the splitting can be done horizontally or vertically, we need to choose one that results in rectangles with better aspect ratios. However, globally searching for the best orientation for each node over time is time consuming. Hence, we use a heuristic method that uses the median weight of the time series of each node to split $R$ in both orientations and then choose one that results in better aspect ratios. In doing so, a global layout tree $LT$ is obtained, see an example in Fig. 2(c).

**Computing Layout.** Following the same splitting strategy in $LT$, we compute a treemap for each time step by using the corresponding data values to determine all splitting positions. In doing so, the resulted treemaps are highly stable, while maintaining good aspect ratios. Fig. 2(d) shows four exemplar treemaps created in this way.

To further improve the visual quality, we allow users to perform a flip operation to leaf nodes at each time step. Specifically, for each node at the second last level, we compare the aspect ratios $\alpha_o$ and $\alpha_n$ resulted by the given orientation and the other one and select the other orientation if $\alpha_o/\alpha_n$ is larger than a threshold. In our experiments, we found that setting the threshold to 1.1 works well for most data. The flip operation is optional; we refer to our method without flipping as SizePairs (SP) and with flipping as SizePairs-F (SP-F) hereafter.

Since our method allows node size to be zero during pairing, it does not require any special strategy to handle insertions, deletions or reinsertions. In contrast, LM has to find a new position for the inserted nodes by optimizing the aspect ratios, which leads to instability. As shown in Fig. 6, our method ensures that nodes ($C$) are reinserted at a consistent place after being deleted.

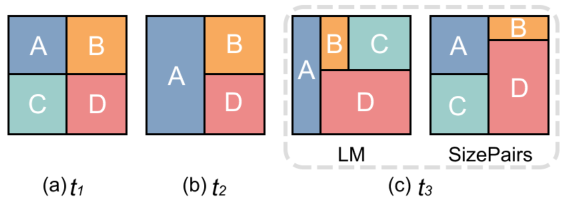


Fig. 6. Illustrating the difference between LM and our method in handling insertions.

## 3.3 Customizing Global Layout Tree

Since our method separates the global layout tree construction from the final treemap generation, users are allowed to interactively edit the layout tree for meeting application requirements. Here, we provide two strategies for customizing the layout tree.

**Stable Node Re-pairing.** Given a treemap, it is hard for users to compare the sizes of two non-adjacent rectangles of interest. To overcome this drawback, SizePairs allows users to interactively re-pair such nodes by exchanging the corresponding leaf nodes. An example is shown in Fig. 7(a), where the node $R_1$ and $L_2$ are exchanged in the tree (see top left), resulting in a new treemap (see bottom right) with $R_1$ and $R_2$ placed together. It should be noted that the gray nodes in Fig. 7(a) are arranged at the same or similar positions since our global layout tree ensures the layout stability.

**Label Friendly Splitting.** The interactive exploration often involves examining the labels of some leaf nodes. However, the splitting orientation results in the rectangles with good aspect ratios, yet which might not fit with the label size. For such rectangles, SizePairs alleviates the issue by changing the splitting orientation of its parent node to see if the label can be clearly shown. Fig. 7(b) shows an example, where the labels with red and blue on top cannot be discerned but are nicely placed after changing the splitting orientations.

## 4 Evaluation

In this section, we present a benchmark that quantitatively and qualitatively compare our method with the state-of-the-art methods in terms of visual quality, short- and long-term stability, and computation speed.

**Measures.** As mentioned above, we used three metrics to evaluate the quality of treemap layouts: (1) corner-travel distance for measuring short-term stability, (2) normalized location drift for long-term stability, and (3) aspect ratio for visual quality.

Corner-travel distance is a common metric for measuring the location change of the four corners of a rectangle between two layouts, $L$ and $L'$. Let $R_i$ ($R'$) denote the rectangle in $L$ ($L'$), and $p_i$, $q_i$, $r_i$, $s_i$ ($p'_i$,$q'_i$,$r'_i$,$s'_i$)

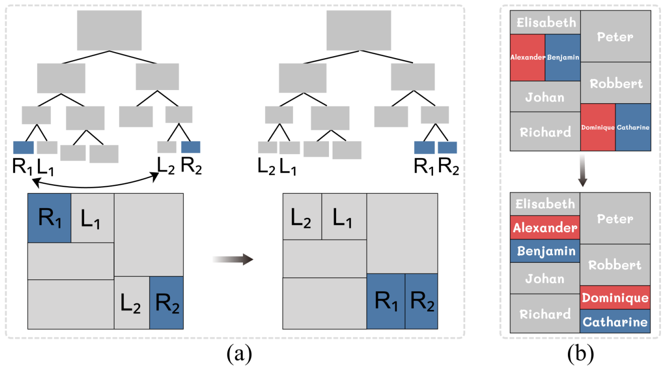


Fig. 7. (a) Interactively re-pairing the rectangles corresponding to leaf nodes $R_1$ and $R_2$ in the global layout tree, resulting in a desired treemap; (b) changing the splitting orientations in the global layout tree for efficiently place labels.

the coordinates of the four corners of $R_i$ ($R'_i$), respectively. The corner-travel distance between $R_i$ and $R'_i$ is defined as follows [28]:

$$CT(R_i, R_i') = \frac{||p_i - p_i'||_1 + ||q_i - q_i'||_1 + ||r_i - r_i'||_1 + ||s_i - s_i'||_1}{4\sqrt{w(R)^2 + h(R)^2}} \quad (4)$$

where $||x||_1$ denotes the $l_1$ norm, and $w(R)$ and $h(R)$ denote the width and height of the entire layout, respectively.

The corner-travel distance (Eq. 4), however, does not consider data changes between $L$ and $L'$ that produce inevitable layout changes. Therefore, if the weight of $R_i$ changes a lot, $CT(R_i, R'_i)$ tends to overestimate the loss in stability since it also captures inevitable size changes. To compute the stability between two layouts more accurately, Vernnier et al. [28] measured the corner-travel distance between $L^*$ and $L'$ instead of $L$ and $L'$ where $L^*$ is a layout that is closest to $L$ but built on the weights of $L'$. We also followed this practice, the details on generating $L^*$ from $L'$ and $L$ can be found in [28].

Due to the normalization of the corner-travel distance by definition, it lies in the range $[0,1]$. Since there are multiple rectangles in a layout, we compute the unweighted mean of $CT(R_i^*, R'_i)$ for each $R_i$ that is present in both layouts, $L^*$ and $L'$. A smaller value is preferred for the corner-travel distance; a value of 0 means the two layouts match perfectly (i.e., the layout is completely stable).

In addition to the corner-travel distance, we used the normalized local drift as an indicator for long-term stability [26]. Since the corner-travel distance only considers two layouts at adjacent time steps, location shifts made throughout multiple time steps are not captured well. For example, if a rectangle gradually moves from the top-left to the top-right corner of a layout over time, the corner-travel distance will only capture the difference between two consecutive layouts, which would be relatively small on average. However, from a global point of view, the first layout would be completely different from the last.

To penalize such cases, we also computed the normalized location drift for each method. For a rectangle $R_i$ in time steps $[1,2,\cdots,m]$, let $C_{i,j}$ be the center of the rectangle at time step $j \in [1,m]$. The center of gravity for $R_i$ throughout the time steps, $CoG(R_i)$, is defined as the mean of $C_{i,j}$ for every $j$. Then, the normalized location drift of $R_i$ is defined as follows:

$$NLD(R_i) = \frac{1}{\sqrt{w(R)^2 + h(R)^2}} \cdot \frac{1}{m} \cdot \sum_{j=1}^{m} || CoG(R_i) - C_{i,j}||_2 \quad (5)$$

where $||x||_2$ denotes the $l_2$ norm. Location drift was originally introduced in [26], but we further divide the value by the diagonal length of the entire layout to normalize it into a range $[0,1]$. We computed the mean of $NLD(R_i)$ for every rectangle $R_i$ in the dataset. Note that there can be rectangles that appear or disappear in the middle of a time sequence since the dataset is dynamic, which was not covered in the original paper [26]. To reflect those changes in location drift more accurately, we computed the weighted mean of location drifts, giving a weight to $NLD(R_i)$ proportional to the number of time steps in which $R_i$ was present. A smaller value for the location drift is desired.

The mean aspect ratio of rectangles has served as a metric for visual quality, as rectangles with extreme aspect ratios, e.g., elongated rectangles, are hard to perceive. The aspect ratio of a rectangle $R_i$ is defined as follows [28]:

$$AR(R_i) = \frac{min(w(R_i), h(R_i))}{max(w(R_i), h(R_i))} \tag{6}$$

where $w(R_i)$ and $h(R_i)$ are the width and height of $R_i$. As explained in [28], $AR(\cdot)$ is the reciprocal of the usual definition for the aspect ratio. This allows $AR(\cdot)$ to be bounded in $[0,1]$, making the mean aspect ratio more robust. For this measure, all rectangles are weighted equally.

Traditionally, an aspect ratio to 1 has been considered ideal, but a user study [16] revealed that humans are more accurate in comparing size differences between rectangles of a 2/3 aspect ratio than squares. Although the optimum aspect ratio for size comparison is still under investigation, a range $[2/3, 1]$ seems reasonable for the desired aspect ratio. We found out that, in practice, it is even challenging to achieve the lower bound of the range (i.e., 2/3) since the dataset is dynamic. Therefore, we consider a higher AR is better.

**Datasets.** As benchmark datasets we used a collection of time-dependent hierarchical datasets proposed by Vernier et al. [28] with 2,405 datasets in total. They characterized each dataset in the collection in terms of four features, assigning them to four subclasses. Below are the features and subclasses they used:

- Levels of hierarchy: 1 level (1L), 2 or 3 levels (2/3L), and 4 or more levels (4+L)
- Variance of node weights: low (LWV) and high (HWV)
- Speed of weight change: low (LWC), regular (RWC), and spiky (SWC)
- Insertions and deletions: low (LID), regular (RID), and spiky (SID)

Note that there is a different amount of datasets for each combination of subclasses. Therefore, we averaged the three metrics over datasets in the same combination. For more details on classification, please refer to the original paper [28].

**Methods.** As mentioned above, we compared SizePairs **(SP)** with LM [25] and GIT [29] since they exhibit a good trade-off between stability and visual quality [28]. We did not include methods that were optimized for a certain aspect; for example, Slice-and-Dice [23] generates very stable layouts but their visual quality (i.e., mean aspect ratio) was about three times worse than that of LM or GIT [28]. On the other hand, unordered methods such as Squarified treemaps [3] often achieve near-optimal aspect ratios but they do not consider the stability between layouts at all. Therefore, we excluded such "extreme" methods from our evaluation.

As a result, we compared the following five methods with specific parameter settings: **LM0**, **LM4**, **SP**, **SP-F**, and **GIT**. LM0 and LM4 are the incremental algorithm introduced in [25] with no local moves allowed (LM0) and with four local moves allowed (LM4). These methods were also included in the previous evaluation by Vernier et al. [28]. We included two versions of our method with different settings: SizePairs without flipping **(SP)** and with flipping **(SP-F)**. Note that we also included Hilbert and Moore treemaps (**HIL**) for measuring long-term stability as this method stems from the paper where location drift was first introduced [26].

**Parameter Choices.** Among the tested methods, GIT and HIL do not have any parameter, while LM and our method have a parameter: the maximal number of moves and $\omega$. For LM, we follow the recommendation from its authors to set these parameters to 0 and 4, respectively. For our method, prior to the comparison, we investigated the effect of hyperparameter $\omega$ on visual quality and stability. We randomly chose 100 datasets in our corpus and computed their corner-travel distance, location drift and aspect ratio, while changing $\omega$ from 0 to 1. As shown in Fig. 8, when $\omega$ is high, the cost term (Eq. 3) puts more emphasis on the compensation degree, favoring stable layouts; two curves for stability, the yellow curve in Fig. 8(a) and the curve in Fig. 8(b), decrease as $\omega$ increases. When $\omega$ is low, the stability measures degrade but the gain from aspect ratio is not substantial. In summary, a value in the range $[0.4, 0.8]$ seems a reasonable choice for $\omega$; therefore, we set $\omega$ to 0.5 in our experiment (highlighted as dotted lines in Fig. 8).

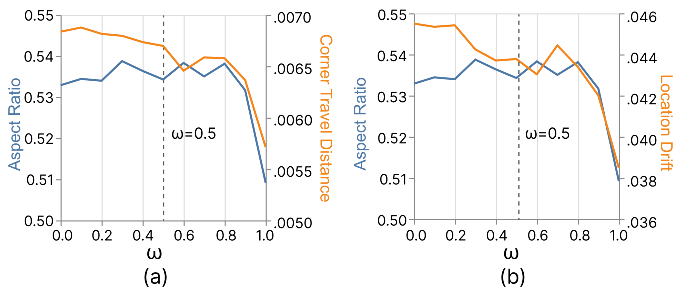


Fig. 8. Effect of hyperparameter $\omega$ on visual quality (aspect ratio) and stability (corner-travel distance and location drift). For the aspect ratio, a higher value is better. For corner-travel distance and location drift, a lower value is better.

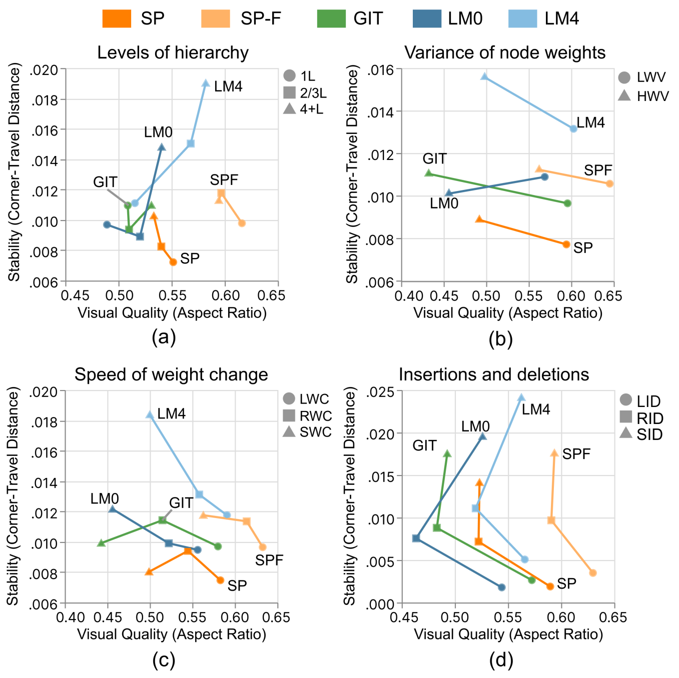


Fig. 9. Relationship between visual quality (aspect ratio) and short-term stability (corner-travel distance) of different methods for datasets grouped by their characteristics. Note that for aspect ratio, a higher value is better, and for stability, a lower value is better. Therefore, a method is superior to another if its point is closer to the bottom-right corner.

**Quantitative Results.** We first reproduced previous benchmark results (Fig. 9) with our methods, SP and SP-F, included. Each plot in Fig. 9 shows the relationship between mean corner-travel distance (i.e., short-term stability) and mean aspect ratio (i.e., visual quality) for different methods with datasets grouped by subclasses in a certain feature. The position of a point in the plot represents the mean corner-travel distance and mean aspect ratio with its shape encoding one subclass. We colorize and connect points using polylines for the same method consistently. Recall that for the corner-travel distance, a lower value is

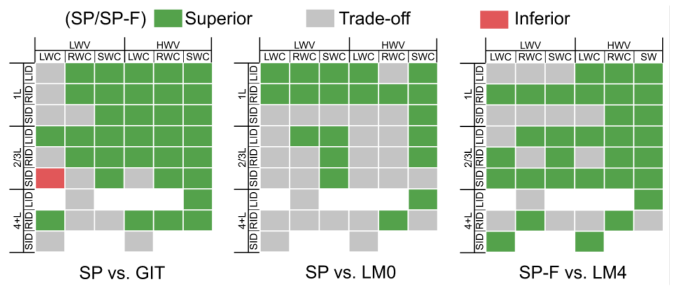


Fig. 10. Superiority of SP(F) over LM and GIT. Green cells indicate SP(F) outperforms GIT or LM on average in terms of corner-travel distance and aspect ratio for datasets with a certain feature combination. Gray means there is a trade-off (e.g., aspect ratio is better but stability is worse). Red means our method was inferior in terms of both metrics.

better while for the aspect ratio, a higher value is better. Therefore, an optimal method, if existing, will be located at the bottom-right corner of the plots.

Overall, we could reconfirm the trade-off between stability and visual quality. The trade-off was clearer when comparing two versions of the same method (e.g., SP vs SP-F). For example, for all subclasses, SP generated more stable layouts than SP-F while SP-F generated layouts with better aspect ratios. This was also true for LM; LM0 was always more stable than LM4 while LM4 exhibited a better visual quality than LM0. Due to these trade-offs, we could not decide globally on the best version for each method.

To perform the comparison between different methods, we decided to make pairwise comparison between the methods that exhibited similar trade-offs. For example, SP, LM0, and GIT commonly favored stability over visual quality, while SP-F and LM4 put more weights to the aspect ratio. Thus, we split methods into two groups and compare methods within the same group.

Fig. 9 shows that, for most subclasses, SP was superior to LM0 (81.8%, 9/11) and GIT (90.9%, 10/11) in terms of stability and visual quality (the points for SP are located closer to the bottom-right corner than those for LM0 and GIT). SP-F outperformed LM4 in terms of both metrics for all subclasses (100%, 11/11).

To further investigate under which dataset conditions SP and SP-F outperform LM and GIT, we compared their performance for every combination of subclasses in Fig. 10. The first table in Fig. 10 compares SP and GIT; it shows all possible combinations of subclasses for the four features, and a cell is colored in green if SP dominates GIT in terms of both measures, gray if there is a trade-off, red if GIT dominates SP, or white if there was no dataset under the corresponding subclass combination. In summary: SP outperforms LM0 in 20 cases (43.5%, 20/46) and GIT in 33 cases (71.7%, 33/46). SP-F outperforms LM4 in 31 cases (67.4%, 31/46). There is no case under which LM0 and LM4 dominate SP and SP-F, but there is only one combination where GIT outperforms SP. We further inspected this case and found that the datasets with this combination have weights that almost do not change over time. In such a case, GIT can maintain the near-optimal aspect ratio from the initial layout.

Fig. 11 shows the normalized location drift (i.e., long-term stability) of the methods for different feature subclasses. In this comparison, we included the Hilbert and Moore treemaps (**HIL**). Overall, SP and GIT showed the best stability, followed by SP-F. In 7 out of 11 cases (63.3%), SP showed the lowest normalized location drift; in the remaining 4 cases, GIT was the best. LM0 and LM4 were better than HIL but underperformed SP and GIT.

**Qualitative Results.** To understand how well each method preserves the stability of layouts, we inspected the layouts of a single-level hierarchy and a multi-level hierarchy. For the single-level hierarchy, we used the WorldBankHIV dataset that consists of 107 nodes and 25 time steps. Fig. 1 shows the corresponding treemaps . At $t = 0$, all methods generate layouts with good aspect ratios. However, in the layouts made by HIL (Fig. 1(a)), the topology between rectangles, which is depicted

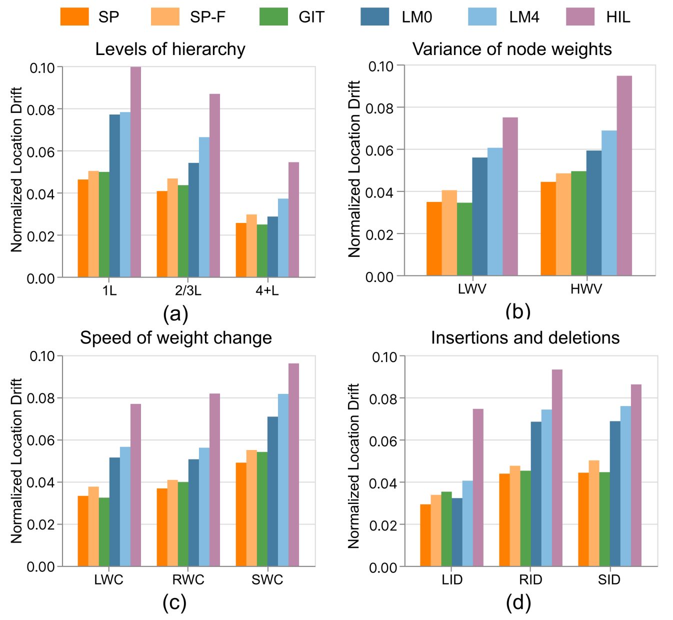


Fig. 11. Normalized location drift of different methods: A lower value is better. SP exhibits the lowest drift in 7 out of the 11 subclasses. Even SP-F that puts more emphasis on visual quality shows comparable or better stability than previous stable methods, such as GIT, LM, and HIL.

as similarity of colors, was lost at $t = 10$ and $t = 20$. It was better preserved in the layouts made by stable methods such as LM4 and GIT (Figs. 1(b,c)), but their visual quality degrades as $t$ increases, as shown by the squeezed rectangles marked with squares and triangles at $t = 20$. In contrast, both, stability and visual quality, were well maintained by SP and SP-F layouts throughout all time steps. See how the displacement between the rectangles embellished with symbols (circle, square, and triangle) is preserved by our SP and SP-F layouts.

To compare the layouts for a multi-level hierarchy, we visualized the NetMigration dataset that includes the top 100 countries with the highest number of emigrants ($n = 202$ and $m = 11$) from 2008 to 2018 (Fig. 12). It has a two-level hierarchy, continent and country. Similar to the WorldBankHIV dataset, LM4 and GIT layouts start with a good aspect ratio, but the visual quality degrades in later frames. For example, in the layouts for the last time step (2018), there are many elongated rectangles with extreme aspect ratios, shown as a "barcode" pattern in the layout (see the last layout of LM4). After carefully checking the results, we found that such patterns appear due to datasets being highly variable in weights, whereas only SizePairs can adapt for them.

With the NetMigration dataset, we could also observe how the methods deal with insertions and deletions. Since the dataset only listed the top 100 countries for each year, some disappear and later re-appear in the list. We found six countries (colored rectangles in Fig. 12) that disappeared in 2012 and re-appeared in 2013. In the LM4 and GIT layouts, such re-appearing countries were placed to a new location that was distant from the position they were in 2011 (e.g., blue rectangles in 2011 and 2013). In contrast, in the SP-F layouts they were placed to locations similar to 2011, making the comparison between 2011 and 2013 easier. This is because SP treats deleted nodes as zero-sized nodes in the layout tree and does not completely detach them from the tree. Later it places the nodes to similar locations when they re-appear.

In both datasets, we observed a degradation in visual quality for LM and GIT layouts. We further investigate these phenomena by plotting the changes for the mean aspect ratio over time for the two datasets Fig. 13. The actual treemap layouts for these datasets can be found in Fig. 1 and Fig. 12. As shown in Fig. 13, GIT and LM layouts usually start with an aspect ratio close to 1. This is because they construct an initial layout optimized for the first time step. However, as time (on the $x$-axis) progresses, the aspect ratio gradually decreases and eventually

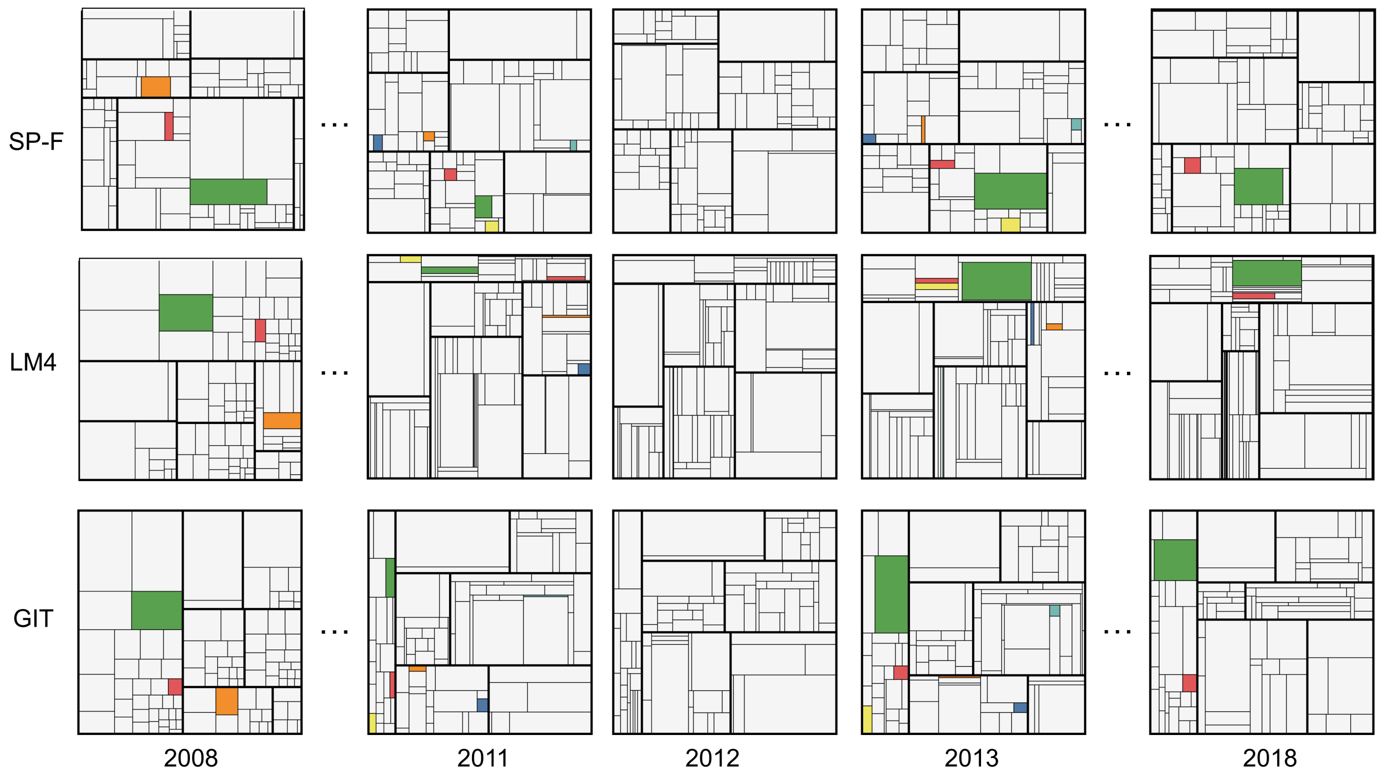


Fig. 12. Treemaps for the NetMigration dataset generated by SP-F, LM4, and GIT. Nodes that disappear in 2012 and re-appear in 2013 are placed consistently to their previous locations by SP-F.

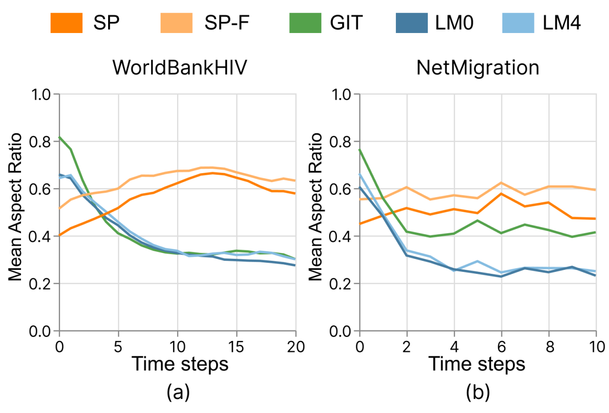


Fig. 13. Mean aspect ratio over time. For GIT and LM, the mean gradually degrades due to accumulating weight changes.

Table 1. Average running time of different methods for computing the layouts for various datasets.

| Dataset | Size | | | Running Time (*ms*) | | | | |
|---|---|---|---|---|---|---|---|---|
| | *n* | *m* | lv. | **SP** | **SP-F** | **GIT** | **LM0** | **LM4** |
| SafeSanitation | 82 | 17 | 1 | **39** | 40 | 136 | 58 | 2,005 |
| Coffee | 86 | 20 | 3 | **22** | 29 | 136 | 34 | 277 |
| LaborEducation | 89 | 29 | 1 | **143** | 153 | 206 | 688 | 1,727 |
| WorldBankHIV | 107 | 25 | 1 | 112 | 113 | 278 | **69** | 6,275 |
| FoodDeficit | 116 | 26 | 2 | 68 | 74 | 219 | **47** | 1,866 |
| BoundRate | 145 | 23 | 2 | **72** | 74 | 242 | 265 | 1,583 |
| MaternalDeaths | 182 | 27 | 2 | **78** | 82 | 359 | 79 | 4,692 |
| Population | 217 | 59 | 2 | **173** | 175 | 704 | 196 | 16,977 |
| Names | 372 | 24 | 1 | 2,042 | 2,284 | **1,598** | 12,210 | 32,434 |
| TMBDChildren | 670 | 20 | 2 | **128** | 130 | 5,491 | 953 | 6,031 |

converges to a very poor ratio close to 0.3. This may result from the fact that these methods locally fix the previous layout for the current time step, not considering the entire time series, so distortions accumulate due to weight changes. Such a drop in visual quality can considerably hinder an analysis for later time steps. In contrast, SP shows a more consistent visual quality. The reasons may be that 1) SP constructs an initial layout considering the median weights over all time steps and 2) the size difference term (Eq. 2) makes the layout more robust to weight changes by matching similar sized nodes.

**Computation Time.** We compared the efficiency of the methods by measuring their average running time for computing the layouts for 10 datasets. We used an open-source implementation of LM and GIT [5], which was also used in the previous benchmark [28]. The implementation of SP and SP-F is available in our repository on GitHub[1]. All methods are written in Java, and the experiment was performed on a desktop with Intel(R) Core i7-9700F CPU@3.00GHz and 16 GB of RAM. We ran each method on each dataset ten times and calculated the average as the final result. Note that different methods may offer different opportunities for acceleration, e.g., parallelization, so when they are fully optimized, the ranking between them can change. For example, the computation of the cost function (Eq. 3) in SP can be fully parallelized between node pairs since there is no data dependence. In our comparison, however, neither method used a specific acceleration technique.

The result is shown in Table 1. We chose the 10 datasets with different numbers of nodes and time steps: *n* indicates the maximum number of nodes included in a layout, *m* is the number of time steps in the data. Note that the number of nodes in each layout varies due to insertions and deletions. Overall, LM had a long running time compared to other methods. LM4 was the slowest followed by LM0. As an example, LM4 took more than 30 seconds to generate the layouts for 24 time steps in the Names dataset.

[1] Anonymized for review

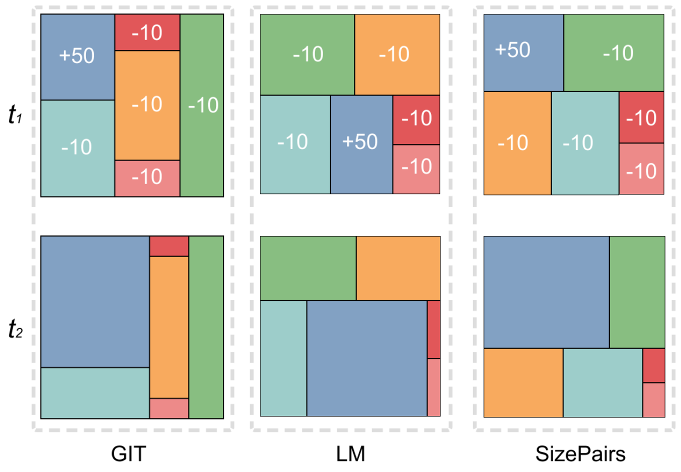


Fig. 14. A challenging case where no pair of nodes is mutually able to complement for size changes over two time steps. The numbers in each rectangle at $t_1$ indicate the amount of size changes at $t_2$.

There was not much difference between SP and SP-F, both methods showed a comparable running time to GIT. SP and SP-F slowed down for large numbers of nodes in one level. For example, although TMBD-Children has more nodes (670 nodes), SP and SP-F took about six times longer to compute the layouts for Names (372 nodes) since all nodes were at the same level. This is due to the fact that when a hierarchy is present in data, the number of nodes (i.e., siblings) to be considered in one hierarchical pairing operation is reduced, which is the main bottleneck of our method. As a result, GIT was faster than SP and SP-F for datasets without hierarchies (e.g., SafeSanitation, LaborEducation, WorldBankHIV, and Names) while SP and SP-F were faster for datasets with hierarchies (the remaining 6 datasets in Table 1).

**Challenging Cases.** In some cases, there are no data items that can complement for size changes of other nodes over time. Fig. 14 shows an example dataset consisting of six nodes and two time steps, where one node's weight increases by 50 and the other five's decrease by 10. Thanks to the size difference term in our cost function (see Eq. 3), SizePairs chooses the nodes with similar sizes in the pairing process. As a result, it yields better aspect ratios than other methods while maintaining similar stability.

**Summary.** In summary, through the benchmark tests and our qualitative investigation we found that:

- Despite the known compromise between stability and visual quality, SizePairs offers better trade-offs, often outperforming previous methods such as LM, GIT, and HIL in terms of corner-travel distance, mean aspect ratio, and normalized location drift (**DG1** and **DG2**);
- As a result, SizePairs better preserves the topology between nodes and their aspect ratios throughout the entire time period, while other methods often distort them especially at later time steps; and
- The running time of SizePairs is faster than LM and comparable to GIT, showing the best performance among state-aware treemapping methods, at least when a hierarchy is present in data (**DG3**).

## 5 Discussion & Future Work

As shown in the Evaluation section, SizePairs, a new temporal treemap, outperforms state-of-the-art methods in generating layouts of high visual quality and strong stability, while being faster than the local moves method. Unlike previous methods, it constructs a global layout tree by considering the entire time series. Specifically, SizePairs employs a new hierarchical size-based pairing algorithm that recursively merges pairs of nodes with mutually compensating changes over time and similar sizes. SizePairs further improves visual quality and stability by optimizing the splitting orientation of each internal node. Based on this global layout tree, the treemap layout of each time step can be generated rapidly, while allowing users to interactively customize treemap layouts for meeting application requirements in a stable way.

As non-treemap-based alternatives, a few hierarchical data visualization techniques [2, 17, 19] can visualize hierarchical changes over time in a static overview by showing each tree node as a stream. However, these techniques are limited to a subset of hierarchical changes in the data [17, 19]. Although Splitstreams [2] support all possible hierarchical changes, all stream-based techniques have two inherent drawbacks: one is not scalable to hundreds of nodes and the other is suffering from a large number of stream crossings for complex data. In contrast, temporal treemaps, especially SizePairs, do not have these issues.

However, our approach also has some limitations. First, SizePairs might not be the best method for data with small changes over time, where our hierarchical pairing algorithm cannot find proper nodes to merge. One example is the data corresponding to the single red cell in Fig. 10. In such cases, GIT and LM both are better choices, since they generate a high quality treemap layout for the first time step. We like to quantify data characteristics and use this information to automatically suggest a proper treemap layout algorithm for different datasets. Second, SizePairs is built on the entire time series, which limits its usefulness in dealing with the streaming data. To address this issue, we will investigate the possibility of combining incremental hierarchical clustering algorithms [21] with SizePairs for dynamically updating the global layout tree. Last, it is unclear if the trade-off between visual quality and stability made by SizePairs better supports data analysis tasks. Therefore, it would be useful to conduct a user study to better understand the trade-off and use it to further improvements of the layout algorithm.

## Acknowledgments

This work was supported by the grants of the NSFC (62132017, 62141217), and the grants of National Research Foundation of Korea (No. NRF-2021R1F1A1055995).

## References

[1] B. B. Bederson, B. Shneiderman, and M. Wattenberg. Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies. *ACM Transactions on Graphics*, 21(4):833–854, 2002. doi: 10.1145/571647.571649

[2] F. Bolte, M. Nourani, E. D. Ragan, and S. Bruckner. Splitstreams: A visual metaphor for evolving hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 27(8):3571–3584, 2020. doi: 10.1109/TVCG.2020.2973564

[3] M. Bruls, K. Huizing, and J. J. v. Wijk. Squarified treemaps. In *Joint Eurographics and IEEE TCVG Symposium on Visualization*, pp. 33–42, 1999. doi: 10.1007/978-3-7091-6783-0_4

[4] C. Bu, Q. Zhang, Q. Wang, J. Zhang, M. Sedlmair, O. Deussen, and Y. Wang. Sinestream: Improving the readability of streamgraphs by minimizing sine illusion effects. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1634–1643, 2020. doi: 10.1109/TVCG.2020.3030404

[5] A. cluster of TU Eindhoven. Treemap comparison, 2020.

[6] E. Cuenca, A. Sallaberry, F. Y. Wang, and P. Poncelet. Multistream: A multiresolution streamgraph approach to explore hierarchical time series. *IEEE Transactions on Visualization and Computer Graphics*, 24(12):3160–3173, 2018. doi: 10.1109/TVCG.2018.2796591

[7] B. Engdahl. Ordered and unordered treemap algorithms and their applications on handheld devices. *Master's degree project, Department of Numerical Analysis and Computer Science, Stockholm Royal Institute of Technology, SE-100*, 44(3), 2005. doi: 10.1.1.103.8749

[8] D. Eppstein, E. Mumford, B. Speckmann, and K. Verbeek. Area-universal and constrained rectangular layouts. *SIAM Journal on Computing*, 41(3):537–564, 2012. doi: 10.1137/110834032

[9] S. Hahn. Comparing the layout stability of treemap algorithms. In *Proceedings of the HPI research school on service-oriented systems engineering*, pp. 71–79, 2015.

[10] S. Hahn, J. Bethge, and J. Döllner. Relative direction change-a topology-based metric for layout stability in treemaps. In *Proceedings of the International Conference on Information Visualization Theory and Applications*, pp. 88–95, 2017. doi: 10.5220/0006117500880095

[11] S. Hahn, J. Trümper, D. Moritz, and J. Döllner. Visualization of varying hierarchies by stable layout of voronoi treemaps. In *Proceedings of the International Conference on Information Visualization Theory and Applications*, pp. 50–58, 2014. doi: 10.5220/0004686200500058

[12] S. Havre, E. Hetzler, P. Whitney, and L. Nowell. Themeriver: Visualizing thematic changes in large document collections. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):9–20, 2002. doi: 10.1109/2945.981848

[13] J. Heer and M. Bostock. Crowdsourcing graphical perception: using mechanical turk to assess visualization design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 203–212, 2010. doi: 10.1145/1753326.1753357

[14] D. Holten, R. Vliegen, and J. J. Van Wijk. Visual realism for the visualization of software metrics. In *Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pp. 27–32. IEEE, 2005. doi: 10.1109/VISSOF.2005.1684299

[15] B. Johnson and B. Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Proceedings of IEEE Visualization Conference*, pp. 284–291, 1991. doi: 10.1109/VISUAL.1991.175815

[16] N. Kong, J. Heer, and M. Agrawala. Perceptual guidelines for creating rectangular treemaps. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):990–998, 2010. doi: 10.1109/TVCG.2010.186

[17] W. Köpp and T. Weinkauf. Temporal treemaps: Static visualization of evolving trees. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):534–543, 2019. doi: 10.1109/TVCG.2018.2865265

[18] J. Kunkel, C. Schwenger, and J. Ziegler. Newsviz: depicting and controlling preference profiles using interactive treemaps in news recommender systems. In *Proceedings of the ACM Conference on User Modeling, Adaptation and Personalization*, pp. 126–135, 2020. doi: 10.1145/3340631.3394869

[19] J. Lukasczyk, G. Weber, R. Maciejewski, C. Garth, and H. Leitte. Nested tracking graphs. *Computer Graphics Forum*, 36(3):12–22, 2017. doi: 10.1111/cgf.13164

[20] H. Nagamochi and Y. Abe. An approximation algorithm for dissecting a rectangle into rectangles with specified areas. *Discrete applied mathematics*, 155(4):523–537, 2007. doi: 10.1016/j.dam.2006.08.005

[21] N. Sahoo, J. Callan, R. Krishnan, G. Duncan, and R. Padman. Incremental hierarchical clustering of text documents. In *Proceedings of the ACM international conference on Information and knowledge management*, pp. 357–366, 2006. doi: 10.1145/1183614.1183667

[22] W. Scheibel, D. Limberger, and J. Döllner. Survey of treemap layout algorithms. In *Proceedings of the International Symposium on Visual Information Communication and Interaction*, pp. 1–9, 2020. doi: 10.1145/3430036.3430041

[23] B. Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on Graphics*, 11(1):92–99, 1992. doi: 10.1145/102377.115768

[24] B. Shneiderman and M. Wattenberg. Ordered treemap layouts. In *Proceedings of the IEEE Symposium on Information Visualization*, pp. 73–78, 2001. doi: 10.1109/INFVIS.2001.963283

[25] M. Sondag, B. Speckmann, and K. Verbeek. Stable treemaps via local moves. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):729–738, 2017. doi: 10.1109/TVCG.2017.2745140

[26] S. Tak and A. Cockburn. Enhanced spatial stability with hilbert and moore treemaps. *IEEE Transactions on Visualization and Computer Graphics*, 19(1):141–148, 2012. doi: 10.1109/TVCG.2012.108

[27] Y. Tu and H.-W. Shen. Visualizing changes of hierarchical data using treemaps. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1286–1293, 2007. doi: 10.1109/TVCG.2007.70619

[28] E. Vernier, M. Sondag, J. Comba, B. Speckmann, A. Telea, and K. Verbeek. Quantitative comparison of time-dependent treemaps. *Computer Graphics Forum*, 39(3):393–404, 2020. doi: 10.1111/cgf.13989

[29] E. F. Vernier, J. L. D. Comba, and A. C. Telea. A stable greedy insertion treemap algorithm for software evolution visualization. In *Proceedings of the SIBGRAPI Conference on Graphics, Patterns and Images*, pp. 158–165, 2018. doi: 10.1109/SIBGRAPI.2018.00027

[30] M. Wattenberg. Visualizing the stock market. In *CHI'99 extended abstracts on Human factors in computing systems*, pp. 188–189, 1999.