Stable treemaps via local moves

Stable Treemaps via Local Moves

Max Sondag, Bettina Speckmann, and Kevin Verbeek

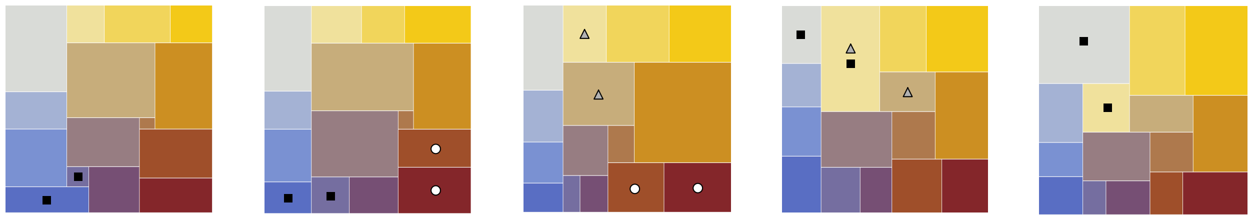

Fig. 1. A *non-sliceable* treemap (a treemap that cannot be recursively sliced into two parts) over time. In each image the weights of the underlying data have changed. To maintain a balance between aspect ratio and stability we modify the treemap via local moves. Symbols (squares, circles, and triangles) mark the pairs of rectangles to which local moves are applied.

Abstract— Treemaps are a popular tool to visualize hierarchical data: items are represented by nested rectangles and the area of each rectangle corresponds to the data being visualized for this item. The visual quality of a treemap is commonly measured via the aspect ratio of the rectangles. If the data changes, then a second important quality criterion is the *stability* of the treemap: how much does the treemap change as the data changes. We present a novel *stable* treemapping algorithm that has high visual quality. Whereas existing treemapping algorithms generally recompute the treemap every time the input changes, our algorithm changes the layout of the treemap using only local modifications. This approach not only gives us direct control over stability, but it also allows us to use a larger set of possible layouts, thus provably resulting in treemaps of higher visual quality compared to existing algorithms. We further prove that we can reach *all* possible treemap layouts using only our local modifications. Furthermore, we introduce a new measure for stability that better captures the relative positions of rectangles. We finally show via experiments on real-world data that our algorithm outperforms existing treemapping algorithms also in practice on either visual quality and/or stability. Our algorithm scores high on stability regardless of whether we use an existing stability measure or our new measure.

Index Terms—Treemap, Stability, Local Moves.

1 Introduction

Treemaps are a well-known and popular tool to visualize hierarchical data. The input for a treemapping algorithm is a set of n data values $a_1, \ldots, a_n$ a hierarchy upon these values (the *tree*), and a shape, most commonly a rectangle R. The data value associated with each interior node in the tree must correspond exactly to the sum of the data values of its children (the values $a_1, \ldots, a_n$ correspond to the leaves of the tree). The output of a treemapping algorithm is a recursive partition of the input shape into disjoint regions such that (a) the size of each region corresponds to its data value, and (b) the regions of the children of an interior node in the tree form a partition of the region of their parent. By definition, treemaps make very efficient use of space.

The vast majority of treemapping algorithms uses rectangles and also this paper focusses exclusively on rectangular treemaps. The visual quality of rectangular treemaps is most commonly measured via the aspect ratio of its rectangles. Shneiderman [13] introduced the first treemapping algorithm ("Slice-and-Dice") in 1991. Despite its popularity it quickly became apparent that Slice-and-Dice was producing rectangles of high aspect ratio and hence poor visual quality. Squarified treemaps by Bruls, Huizing and Van Wijk [3] aimed to ameliorate this fact using a heuristic approach. From a theoretical point of view the aspect ratio of treemaps can become arbitrarily bad: consider a treemap with only two rectangles, one of which has an extremely large area while the other has an extremely small area (and hence necessarily becomes extremely thin). Nevertheless, Nagamochi and Abe [12] describe an algorithm which provably approximates the optimal aspect ratio for a given treemap. Eventually De Berg, Speckmann and Van Der Weele [5] proved that minimizing the aspect ratio for rectangular treemaps is strongly NP-complete.[1]

If the input data change, then a second important quality criterion is the *stability* of the treemap: how much does the treemap change as the data changes. It is clearly desirable that small changes in the data result only in small changes in the treemap. There are a variety of stable treemap algorithms which try to maintain an order on the input data. There are also several related quality metrics which measure how stable a treemap is. We review both in detail in Section 2. All these existing treemap algorithms have two things in common: (i) they recompute the treemap completely when the data changes, and (ii) they use exclusively *sliceable* layouts for their treemaps.

Contribution. We present a novel stable treemapping algorithm that has high visual quality. In contrast to previous treemapping algorithms we (i) adapt our treemap via local modification, and (ii) explore the complete space of possible treemap layouts, including *non-sliceable* layouts (see Fig. 1 for an example of the resulting treemaps). We prove that our approach may result in treemaps of higher visual quality and also show experimentally that our algorithm outperforms existing treemapping algorithms on either visual quality and/or stability.

Definitions and Notation. To describe our contribution in greater detail, we introduce some definitions and notation. First of all, we distinguish *single-level* treemaps and *multi-level* treemaps. A single level treemap has no hierarchy, its tree consist only of a root node with n leaves. Multi-level treemaps correspond to trees with interior nodes in addition to the root. A multi-level treemap has a clear recursive structure: the rectangles which correspond to the children of the root form a single-level treemap (a partition of the input rectangle) and each such rectangle in turn serves as the input rectangle for further

- *Max Sondag is with TU Eindhoven. E-mail: m.f.m.sondag@tue.nl.*
- *Bettina Speckmann is with TU Eindhoven. E-mail: b.speckmann@tue.nl.*
- *Kevin Verbeek is with TU Eindhoven. E-mail: k.a.b.verbeek@tue.nl.*

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

[1]This result was previously claimed in [3] but not in fact proven.

subdivision according to its children. When studying treemaps it is hence generally sufficient to study single-level treemaps, since all results directly extend to multi-level treemaps; a multi-level treemap can be viewed as multiple nested single level treemaps where the input rectangle of the child is the rectangle determined by the parent. This has the added advantage of removing unnecessary complexity from arguments. For all theoretical parts of this paper (Sections 3–5) we hence mostly consider only single-level treemaps. The experimental evaluation in Section 7 uses also multi-level treemaps, which are constructed recursively according to the algorithm which we describe for single-level treemaps in Section 5.

A single-level treemap is a partition of the input rectangle R into a set of n disjoint subrectangles $\mathcal{R} = \{R_1, \ldots, R_n\}$, where each rectangle R_i has area a_i. Such a partition of rectangles into subrectangles is known as a *rectangular layout* L, or *layout* for short. Layouts have been studied in a variety of research areas, including floorplans in architecture and VLSI design and rectangular cartograms in automated cartography. For a layout L, a *maximal segment* is a maximal contiguous horizontal or vertical line segment contained in the union of the boundaries of rectangles in $\mathcal{R}$. We distinguish between two types of layouts: *sliceable layouts* and *non-sliceable layouts*. A layout is sliceable if it can be recursively sliced into two parts along a maximal segment until the layout consists of only a single rectangle. Otherwise the layout is non-sliceable (see Fig. 2 for an example).

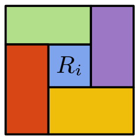

Fig. 2. A non-sliceable layout. Rectangle R_i is the center of a "windmill".

All treemap algorithms discussed in this paper (with the exception of our stable treemapping algorithm) produce only sliceable layouts. This is obvious for Slice-and-Dice, since slicing cuts are an integral part of the algorithm. Squarified treemaps, all algorithms using strips and spirals, and the Pivot-by-* family all explicitly construct slicing cuts. But also the treemaps created using space-filling curves (Hilbert and Moore) are sliceable: the base of the recursion are four rectangles and every layout with 4 rectangles is sliceable.

Organization. Section 2 discusses related work. In Section 3 we first give some additional background on rectangular layouts. We then prove a lower bound on the maximum aspect ratio of sliceable layouts and argue that non-sliceable layouts can achieve a lower aspect ratio. In Section 4 we introduce the *local moves* which we use to locally modify our treemaps. We then prove that these local moves are powerful enough to explore the complete space of treemap layouts – both sliceable and non-sliceable layouts. In Section 5 we present our stable treemapping algorithm using local moves. In Section 6 we introduce a new measure for stability which arguably captures the relative positions of rectangles better than existing stability measures. Finally in Section 7 we report on extensive experiments comparing our new treemapping algorithm to existing treemapping algorithms, both on single and multi-level treemaps using real-world data. The experiments show that our algorithm outperforms existing algorithms also in practice on either visual quality and/or stability. Our algorithm scores high on stability regardless of whether we use an existing stability measure or our new measure.

2 Related work

Shneiderman and Wattenberg [14] proposed the first type of treemap that takes stability into account: the ordered treemap. Here an additional order on the treemap rectangles is specified and rectangles that are near each other in this order are attempted to be placed near each other in the treemap. However, as the input data changes there are no guarantees on how close any two rectangles will stay even if they are neighbors in the order. There are several ordered treemap algorithms: the Pivot-by-(Middle, Size and Split-Size) algorithms by Shneiderman and Wattenberg [14], the Strip algorithm by Bederson, Shneiderman and Wattenberg [2], the Split algorithm by Engdahl [6], the Spiral algorithm by Tu and Shen [16], and the Hilbert and Moore algorithms by Tak and Cockburn [15].

To measure the success of maintaining the order underlying ordered treemaps, Bederson et al. [2] introduced the readability metric. The readability metric measures how often the motion of the reader's eye changes direction as the treemap is scanned in order. In addition, Tu and Shen [16] introduced the continuity metric which measures how often the next item in the order is not the neighbor of the current item. Both these metrics attempt to quantify how easy it is to visually scan an ordered treemap to find a particular item.

To measure the stability of treemaps, Shneiderman and Wattenberg [14] proposed the layout-distance-change function. There are three variations of this function. The first is the variance-distance-change function by Tak and Cockburn [15]. The second variant, also by Tak and Cockburn [15], is locational-drift, which measures the stability over a larger period of time. The final variant, proposed by Hahn et al. [10], measures the stability of non-rectangular treemaps using the distance between centroids of regions. Very recently Hahn et al. [9] proposed to use relative-direction-change, which measures the change in rotation between rectangles. This last measure is related to our proposed stability measure since it incorporates the relative positions of regions. However, we believe that our measure captures the relative position of specifically rectangles better. In Section 6 we discuss stability measures in more detail.

Not all treemaps use rectangles. Alternative models include Voronoi treemaps by Balzer, Deussen, and Lewerentz [1], orthoconvex and L-shaped treemaps by De Berg et al. [5], and Jigsaw treemaps by Wattenberg [18]. Furthermore, Hahn et al. [10] describe an approach for stable Voronoi treemaps.

As mentioned before, rectangular layouts are studied in a variety of research areas. Of particular interest here are several results from VLSI design. In this context two rectangular layouts are considered equivalent if each rectangle has the same adjacencies in both layouts, that is, if the two layouts have the same dual graph. The question then arises how many non-equivalent layouts exist that consist of exactly n rectangles. Yao et al. [19] showed how to represent rectangular layouts with so-called twin binary tree sequences. Using these twin binary tree sequences they proved that the number of non-equivalent sliceable layouts equals the Baxter number [4] and that the number of non-equivalent non-sliceable layouts equals the Schröder number [8]. Using the same representation by twin binary tree sequences, Young, Chu, and Shen [20] showed how to transform any two rectangular layouts into each other using local modifications. Our local moves are inspired by their method. We show in Section 4 how to define simple moves directly on a treemap (and hence avoiding the somewhat involved twin binary trees) to achieve the same result, namely a sequence of simple modifications which transforms any two treemaps into each other.

3 Layouts

For a given set of areas $a_1, \ldots, a_n$ there are multiple ways to draw a treemap, that is, there are multiple layouts that can represent the same set of areas. We want to find the layout that has the highest visual quality, that is, the layout that minimizes the aspect ratios of its rectangles. To do so we need to explore the space of possible layouts. As stated above, all current rectangular treemapping algorithms produce only sliceable layouts. In this section we prove that the quality of treemaps can be improved substantially by considering all possible layouts, sliceable and non-sliceable. Our algorithm, presented in Section 5, is the first treemapping algorithm that can produce all possible layouts.

Two layouts representing different areas cannot be the same. Nonetheless they can have a very similar structure. We therefore consider a combinatorial equivalence between layouts. For a layout L, a *maximal segment* is a maximal contiguous horizontal or vertical line segment contained in the union of the boundaries of rectangles in $\mathcal{R}$. We denote the set of maximal segments of a layout L by $\mathcal{S} = \mathcal{S}(L)$. We define a partial order on maximal segments of the same orientation as follows. For two horizontal maximal segments s_1 and s_2 we say that $s_1 < s_2$ if s_1 is below s_2, and there exists a rectangle in $\mathcal{R}$ that spans from s_1 to s_2. Vertical maximal segments similarly define a partial order from left to right. Following Eppstein *et al.* [7], we say that two layouts L and L' are *order-equivalent* if the partial orders for

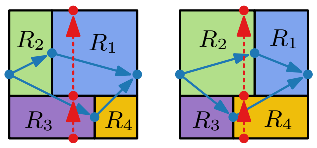

Fig. 3. Two order-equivalent layouts: the blue arrows indicate the partial order on the vertical maximal segments, the red arrows indicate the partial order on the horizontal maximal segments.

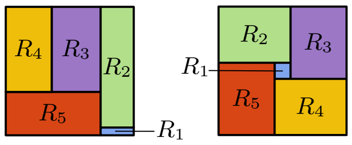

Fig. 5. Let $a_1 = 1$ and $a_2 = a_3 = a_4 = a_5 = 16$. The maximum aspect ratio in the non-sliceable layout on the right is ≈ 1.333. For any sliceable layout the maximum aspect ratio is at least 4.

L and L' are isomorphic.[2] An example of order-equivalent layouts is given in Figure 3. In [7] it was shown that, for any layout L, there is always exactly one layout L' that is order-equivalent to L and correctly represents a given set of areas. Thus, for any fixed set of areas, the possible ways to draw a treemap with these areas corresponds to the set of order-equivalence classes of all possible layouts.

Sliceable and Non-sliceable layouts. If a layout L is sliceable, then all layouts order-equivalent to L are also sliceable. Existing rectangular treemapping algorithms hence exclude a large number of options from consideration, which may result in treemaps of sub-optimal visual quality. We aim to show this formally. Below we prove that, for certain sets of areas, the maximum aspect ratio of any sliceable layout is much larger than the maximum aspect ratio of the optimal layout.

We say that a rectangle $R_i \in \mathcal{R}$ is *grounded* if R_i is bounded by at least one maximal segment s for which it is the only rectangle on that side of s (see Fig. 4). We claim that in a sliceable layout all rectangles are grounded. Indeed, if this is not the case, then there must be a rectangle R_i such that all four bounding maximal segments have at least two rectangles on the side of R_i. This results in a "windmill pattern" with R_i in the center (see Fig. 2). It is not hard to see that any layout that contains a "windmill pattern" is non-sliceable. We can now prove the following theorem.

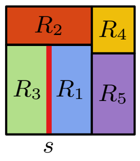

Fig. 4. R_1 is grounded at maximal segment s.

Theorem 1. *The maximum aspect ratio of a sliceable layout L is at least $\sqrt{a_2/a_1}$, where a_1 and a_2 are the smallest and second-smallest area in the layout, respectively.*

Proof. Let ρ be the maximum aspect ratio of L and let R_1 be the rectangle with the smallest area in L. Let rectangle R_2 be adjacent to R_1 such that R_1 is grounded in the maximal segment shared with R_2. Without loss of generality we assume that rectangle R_2 lies to the right of rectangle R_1. Denote the height and width of R_i by h_i and w_i, respectively. From the grounded property we get that $h_1 \geq h_2$. This also implies that $a_2/a_1 \leq w_2/w_1$. From the definition of ρ we get that $h_1 \leq \rho w_1$. We further get that

$$\rho \geq \frac{w_2}{h_2} \geq \frac{w_2}{h_1} \geq \frac{w_2}{\rho w_1}.$$

As a result, $\rho^2 \geq w_2/w_1 \geq a_2/a_1$. Thus the maximum aspect ratio of L is at least $\sqrt{a_2/a_1}$. This is minimized when R_2 is the rectangle with the second smallest area in L. □

Consider the following concrete example with 5 areas: $a_1 = 1$ and $a_2, a_3, a_4, a_5 = 16$. According to Theorem 1 any sliceable layout will have a maximum aspect ratio of at least $\sqrt{16/1} = 4$. On the other hand, there exists a non-sliceable layout with these areas with maximum aspect ratio ≈ 1.333, as is shown in Figure 5. In fact, this difference in maximum aspect ratio between sliceable layouts and non-sliceable layouts can be made arbitrarily large: As a_1 tends to 0, the maximum aspect ratio of the non-sliceable layout in Figure 5 tends to 1, while the maximum aspect ratio of any sliceable layout tends to ∞.

[2] Note that this equivalence is different from the equivalence considered by Yao et al. [19] and mentioned in Section 2: order-equivalent layouts generally do not have the same dual graph.

4 Local moves

Our stable treemapping algorithm uses the concept of *local moves*. A local move changes the order-equivalence class of the layout L by changing the layout L locally. Local moves allow us to traverse between all order-equivalence classes of layouts. Intuitively, we can keep a treemap stable over time by limiting the number of local moves between any two time steps. At the same time, the more local moves we allow, the better the visual quality can be. Local moves hence give us the power to control the tradeoff between stability and visual quality. A local move typically changes the areas of the involved rectangles. We can correct the areas in the resulting layout L' using the hill-climbing algorithm by Eppstein *et al.* [7] (for details see Section 5).

The final layout is order-equivalent to L'. Note that two order-equivalent layouts may have different adjacencies across maximal segments (see Fig. 3). If L' has long maximal segments with many adjacent rectangles on both sides, then adjusting the areas may result in non-local changes of rectangle adjacencies. However, since these changes occur only along maximal segments, they influence the relative positions of rectangles only mildly. We hence claim that our treemaps are stable if we allow only a small number of local moves. The experimental evaluation in Section 7 supports this claim.

Our local moves are inspired by the work of Young et al. [20]. They use a representation of rectangular layouts with twin binary tree sequences and show how to use this representation and an additional labeling to transform any two rectangular layouts into each other using only local moves. Their particular labeling is not suitable for the context of treemaps and the representation by twin binary trees is somewhat cumbersome. Below we hence introduce two new local moves which operate directly on the treemap: *stretch moves* and *flip moves*. We prove that one can transform any two order-equivalence classes of layouts into each other using only these two moves.

Stretch move. Let s be a maximal segment and let R_1 and R_2 be two rectangles adjacent to one of the endpoints of this segment. Without loss of generality we assume that s is a vertical maximal segment. If rectangles R_1 and R_2 do not have the same height we can apply a stretch move. Let rectangle R_2 denote the rectangle with the smallest height. Without loss of generality we assume that rectangle R_1 is to the left of s. To apply the stretch move we then stretch rectangle R_2 to the left over rectangle R_1 as is shown in Figure 6.

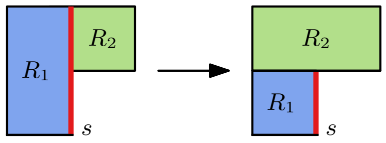

Fig. 6. A stretch move at the upper endpoint of the maximal segment s.

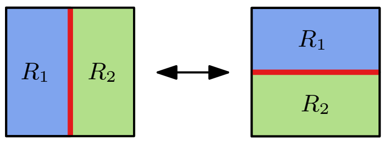

Fig. 7. A flip move on rectangles R_1 and R_2.

Flip move. Let R_1 and R_2 be two rectangles that together form a larger rectangle. The flip move flips the adjacency between R_1 and R_2 from horizontal to vertical or vice versa inside this larger rectangle. An example of a flip move is illustrated in Figure 7.

4.1 Transforming rectangular layouts using local moves

We now prove that we can transform any layout L into any other layout L' using only local moves. For this transformation we need the notion of a *vertical stack layout*. A layout is a vertical stack layout if it has only horizontal (inner) maximal segments. The transformation from L to L' can now be summarized as follows. First we transform L into a vertical stack layout. Next we transform this vertical stack layout into another vertical stack layout. Finally we transform the resulting vertical stack layout into L'. To show the existence of this transformation, we need the three following components.

Transforming a layout to a vertical stack layout. To transform a layout L to a vertical stack layout, we need to eliminate all vertical (inner) maximal segments. Let s be a vertical maximal segment of L. Furthermore, let R_1 be the rectangle adjacent to the left top of s, and let R_2 be the rectangle adjacent to the right top of s. Now first assume that R_1 and R_2 do not have the same height, and assume without loss of generality that the height of R_2 is smaller than the height of R_1. In this case we use a stretch move to stretch R_2 over R_1 (see Fig. 8). If R_1 and R_2 have the same height, then they form a larger rectangle together. We use a flip move on R_1 and R_2 as is shown in Figure 9. Note that, in both cases, we reduce the number of rectangles adjacent to s by at least one. When there are no more rectangles adjacent to s, s will cease to exist. Furthermore, our operations do not introduce new vertical maximal segments. We can thus repeatedly apply this procedure until all vertical maximal segments have been eliminated.

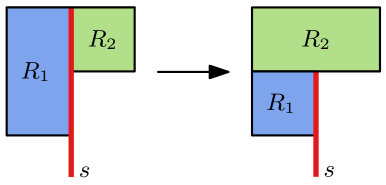

Fig. 8. Rectangle R_2 is stretched over rectangle R_1: s has one less rectangle adjacent to it.

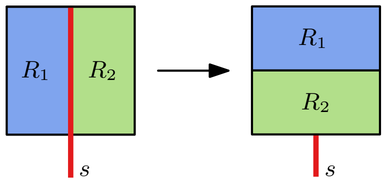

Fig. 9. Rectangle R_1 and R_2 are flipped using a flip move: s has two less rectangles adjacent to it.

Transforming vertical stack layouts. Consider any two adjacent rectangles R_1 and R_2 in a vertical stack layout. We can swap R_1 and R_2 in the vertical stack order by applying two flip moves to R_1 and R_2 (see Fig. 10). Since we can swap any two adjacent rectangles, we can produce any order of rectangles in the vertical stack layout (this process is the same as sorting with *BubbleSort*).

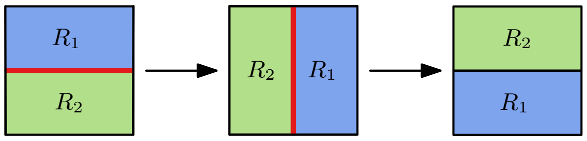

Fig. 10. Rectangles R_1 and R_2 are swapped using two flip moves.

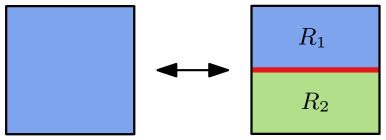

Fig. 11. Inverting the Flip move.

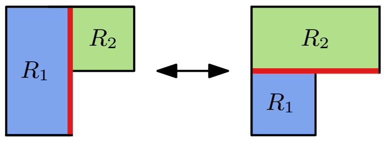

Fig. 12. Inverting the Stretch move.

Inverting local moves. It is easy to see that all local moves can be inverted. Trivially, a flip move is its own inverse (see Fig. 11). Furthermore, a stretch move that stretches R_2 over R_1 can be inverted by a stretch move that stretches R_1 over R_2 (see Fig. 12).

We can now prove the following theorem:

Theorem 2. *For any two layouts L_1 and L_2 with the same set of rectangles, we can transform L_1 into L_2 using only stretch moves and flip moves.*

Proof. We can transform L_1 into a vertical stack layout L_1' as described above. Similarly, we can transform L_2 into a vertical stack layout L_2'. To transform L_1 into L_2, we first transform L_1 into L_1'. Next, we transform L_1' into L_2' using appropriately chosen swaps of adjacent rectangles. Finally we transform L_2' into L_2 by inverting the local moves used to transform L_2 into L_2'. □

We can additionally show that, if the number of rectangles in L_1 and L_2 is n, then we need at most $O(n^2)$ local moves to transform L_1 into L_2.

Please note that the proof above is a so-called "constructive proof of existence". Our argument (i) shows that there always is a set of local moves to transform one layout into the other, and (ii) it describes a way to find these moves. Clearly the resulting transformation is not very natural and we do not intend to use this transformation. Now that we have proven that a transformation always exists, we can find more suitable transformations in practice.

5 Algorithm

We now describe our stable treemapping algorithm *Incremental Treemap* for time-varying data. Our algorithm uses the previous treemap to generate the next one. We therefore need to describe how to transform a treemap T with areas $A = \{a_1, \ldots, a_n\}$ into a treemap T' with areas $A' = \{a_1', \ldots, a_n'\}$.

We first consider only a single-level treemap T. We construct the initial treemap using the approximation algorithm by Nagamochi and Abe [12]. To transform T into T', we use a very simple approach. First we update the treemap T to have the areas in A' using the hill-climbing algorithm by Eppstein *et al.* [7].

The idea of the algorithm by Eppstein *et al.* is as follows. As is shown in [7], there is an induced bijection between the space of coordinates of the maximal segments (*segment space*) and the space of the areas of the rectangles (*area space*). Hence, given a (tangent) vector in the area space, in particular $A' - A$, we can compute the corresponding tangent vector x in the segment space by solving the linear equation $Jx = A' - A$, where J is the Jacobian matrix of the bijection. Thus, we can locally change the areas from A to A' by moving the maximal segments in the direction of x. The Jacobian matrix J is sparse and can easily be computed as, for each rectangle, the area simply depends on the coordinates of the 4 maximal segments bounding the rectangle. We can now proceed as in a gradient descent approach by iteratively changing the maximal segment coordinates by εx, for ε small enough, until we obtain the areas in A'.

Next, we attempt to improve the visual quality of the layout by applying up to d local moves, where d is some predefined small constant (in our experiments $d = 4$). A naive approach would simply try all possible sets of at most d local moves. In Section 5.1 we explain how to choose a suitable subset of possible moves to optimize performance. The areas of the resulting layouts (after the at most d local moves) are then again adjusted using the hill-climbing algorithm by Eppstein *et al.* [7] to generate an order-equivalent layout with the correct areas. We use the layout with the best average aspect ratio to construct T'. Note that we do not change the layout if doing so would lead only to a minor improvement in aspect ratios. Therefore, if L is the layout of T with updated areas A', then we only change L into L' if the sum of aspect ratios in L' is at least some predefined constant c lower than the sum of aspect ratios in L (in our experiments we use $c = 4$).

If T is a multi-level treemap then we use our algorithm recursively on the rectangles that represent subtrees. That is, we first transform the single-level treemap which is formed by the root of T and its children using a set of local moves. Then we recurse into the treemaps inside each of the resulting rectangles R_i and apply the algorithm on the lower levels. The choice of moves is restricted to those moves that involve only subrectangles of R_i which ensures that the hierarchy information is maintained. Clearly changing the layout on a higher level of T has more impact than changing it on a lower level, as it affects all subtreemaps of T. We account for this by adapting the value of c according to the height of the level. In our implementation we are using $c = 4 * \sqrt{\text{height of the level}}$.

Handling additions. When additional data points become available in a changing data set we need to add a new rectangle to the treemap. We introduce such new rectangles before performing any local moves. To add a new rectangle R_k to the treemap, we partition an existing rectangle R_i into two subrectangles R_i and R_k (see Fig. 13). We pick R_i in such a way that the aspect ratios are minimized.

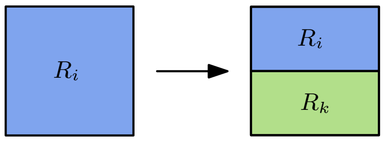

Fig. 13. Rectangle R_k is inserted next to R_i by partitioning R_i into two sub rectangles.

Handling deletions. For similar reasons we may also need to remove a rectangle from the treemap. Removing rectangles is slightly more involved than adding rectangles, and happens before any local moves are performed, but after new rectangles have been added. There are two cases we need to consider when removing a rectangle R_i:

R_i **is grounded:** There necessarily exists a maximal segment s for which R_i is the only rectangle on one side of s. To remove R_i we stretch all rectangles on the other side of s over R_i using stretch moves (see Fig. 14).

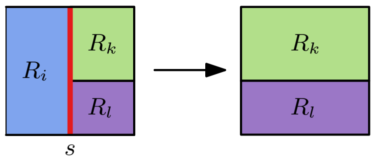

Fig. 14. We delete the grounded rectangle R_1 from the layout by stretching R_2 and R_3 over R_1.

R_i **is not grounded:** R_i must be in the center of a windmill pattern. The goal is now to apply stretch moves to R_i until R_i becomes grounded and we are in the first case. Let e be the edge of R_i that is adjacent to the fewest rectangles on the other side. Since R_i is in the center of a windmill pattern, e must include an endpoint of a maximal segment s. Without loss of generality e is above R_i and the endpoint of s is on the left side of e. Then, as long as there is more than one rectangle on the top side of e, we stretch the leftmost of those rectangles R_j over R_i. As soon as there is only one rectangle R_j on the other side of e, we stretch R_i over R_j (see Fig. 15). If we are unlucky, then R_i is still not grounded, but it is now part of a larger windmill pattern. In that case we repeat the procedure above until R_i finally becomes grounded.

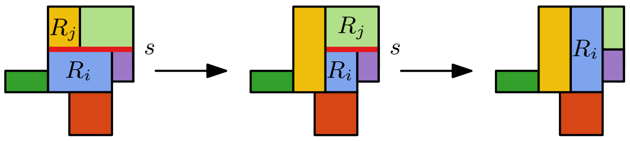

Fig. 15. If rectangle R_1 is adjacent to multiple rectangles of all sides, we can make R_1 a grounded rectangle by repeatedly applying stretch moves over an adjacent maximal segment s.

We summarize our algorithm in the following pseudocode, where $f(L)$ measures the sum of aspect ratios in a layout L.

```
Algorithm IncrementalTreeMap($T$, $A'$, $d$, $c$)
1.  if $T$ is empty
2.      Generate $T'$ using the approximation algorithm.
3.  else
4.      $L$ = correctAreas($L$, $A'$)
5.      Add rectangles to $T$ that need to be in $T'$.
6.      Remove rectangles from $T$ that are not in $T'$.
7.  $\mathcal{Q}_0 = \{L\}$
8.  $L_{best} = L$
9.  for $i = 1$ to $d$
10.     for $L' \in \mathcal{Q}_{i-1}$
11.         for all possible local moves $m$ on $L'$
12.             $L''$ = apply($m$, $L'$)
13.             $L''$ = correctAreas($L''$, $A'$)
14.             if $f(L'') < f(L_{best})$
15.                 $L_{best} = L''$
16.             $\mathcal{Q}_i = \mathcal{Q}_i \cup \{L''\}$
17. if $f(L_{best}) < f(L) - c$
18.     Let the layout of $T'$ be $L_{best}$
19. for all children $T_c$ of $T$
20.     IncrementalTreeMap($T_c$, $A'(T_c)$, $d$, $c$)
```

Finally we give the pseudocode for an implementation of Eppstein's *et al.* [7] algorithm.

```
Algorithm correctAreas($L$, $A'$)
1.  Let $A$ be the areas in $L$.
2.  while $\|A - A'\|$ is not small enough
3.      Let $J$ be the Jacobian matrix of mapping segments to areas.
4.      Solve $Jx = A' - A$ for $x$.
5.      Move maximal segments of $L$ by $\varepsilon x$.
6.      Recompute areas $A$ of $L$.
7.  return $L$
```

5.1 Improving performance

The naive algorithm described above is not very efficient. There are two reasons for that: (i) the number of layouts considered by the algorithm is exponential in d, and (ii) updating the areas using the hill-climbing algorithm is not very efficient. We address these two issues below.

Reducing the number of layouts. We first compute all layouts that are the result of applying one local move. Of these layouts, we only keep the k layouts with the smallest aspect ratios (in our experiments we use $k = 4$). When applying a second local move to one of the k remaining layouts, we consider only those local moves that involve a maximal segment for which the adjacencies have been changed by the first local move. Afterwards, we again keep only the best k layouts and we repeat this procedure until we have applied d local moves per layout. Although this approach may not find the best possible layout, it does perform well in practice and the number of layouts considered is no longer exponential in d.

Updating areas more efficiently. Computing the correct areas for general (possibly non-sliceable) layouts is significantly more difficult than computing the correct areas for sliceable layouts (areas for sliceable layouts can simply be computed recursively). However, most layouts contain large components that are sliceable. We can use this fact to speed up our algorithm. While we can find a maximal segment s that slices the layout, we simply place s according to the areas of the rectangles on its two sides, and continue recursively on both sides of s. When the layout is not sliceable, we try to find maximal segments that have a single rectangle on both sides. These maximal segments can be removed and reinserted later as a slicing maximal segment. Finally, when no such maximal segments remain, we use the hill-climbing algorithm to position the remaining maximal segments. This approach speeds up our algorithm substantially in practice.

6 Measures of Stability

One of the most common measures for stability is the *layout-distance-change* function introduced by Shneiderman and Wattenberg [14]. The layout-distance-change function measures the average change of each rectangle in position and shape between two layouts. There are three variants of this function. The first variant is the *variance-distance-change* function by Tak and Cockburn [15], which measures the variance of the layout-distance-change. This allows us to distinguish between a large number of small changes which might be almost invisible, and a small number of large changes. The second variant is the *centroid-positioning* measure by Hahn et al. [10]. This measure captures the average change of the centroids of rectangles between two layouts. This approach can easily be extended to non-rectangular treemaps. The third and final variant is the *locational-drift* measure, again by Tak and Cockburn [15]. This measure captures how much the rectangles move from their average position over a longer period of time: if rectangles drift around the same position, it is easy to track the rectangles even though the exact position changes every iteration.

All of these measures use the change in absolute position of a rectangle between layouts as their base. We claim that this is not sufficient to measure the stability. We believe that the change in relative position between rectangles is another important factor to determine the stability. If the relative position between a pair of rectangles R_2 and R_3 is unchanged, then even if the absolute positions of R_2 and R_3 change drastically, it is not too difficult to keep track of R_2 and R_3. For example, in Figure 16 rectangle R_1 has changed positions with rectangles $R_2, \ldots, R_5$. In absolute distance, the layout has changed significantly and thus layout-distance-change and its variations give a high score. However, as all rectangles $R_2, \ldots, R_5$ have maintained their relative position with regard to each other, it is actually quite easy to track the rectangles from one layout to the next. By finding a single rectangle from $R_2, \ldots, R_5$ we can easily find all other rectangles from $R_2, \ldots, R_5$. Thus, when only small changes in the relative position of the rectangles occur, the layout seems to be relatively stable even when the absolute positions change drastically.

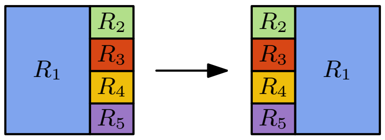

Fig. 16. Rectangle R_1 has swapped with rectangle $R_2, \ldots, R_5$. Even though the absolute positions have significantly changed, it is still easy to track the changes.

6.1 A new measure of stability

We introduce a new measure for stability that aims to properly capture the change in relative position between rectangles. The relative position between two rectangles is often perceived as above or below, and left or right. To measure the relative position with respect to a rectangle R_i, we subdivide the space around R_i into 8 sections $\{S_1, S_2, ..., S_8\} = \mathcal{S}(R_i)$ by extending the sides of R_i (see Fig. 17). Section S_1 represents the East, Section S_2 the NorthEast, etc. The relative position of R_j with respect to R_i is now determined by the percentage of R_j in each of the sections of $\mathcal{S}(R_i)$.

We determine the change of the relative position between rectangles R_i and R_j in layouts L and L' by calculating to what degree rectangle R_j stays in the same sections of rectangle R_i. Let $p^k_{ij}(L)$ be the percentage of R_j that is in section S_k of $\mathcal{S}(R_i)$ in layout L. We define the *relative-position-change* between R_i and R_j as follows:

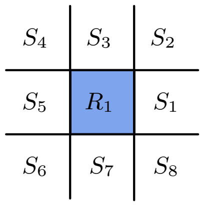

Fig. 17. The space around rectangle R_1 is subdivided into 8 sections.

$$D^{rel}_{ij}(L, L') = \frac{1}{2} \sum_{k=1}^{8} \left| p^k_{ij}(L) - p^k_{ij}(L') \right| \qquad (1)$$

For example, in Figure 18 rectangle R_2 was for 25% in S_1 and for 75% in S_2 of rectangle R_1. After the change R_2 is for 100% in S_1 which results in a score of 0.25.

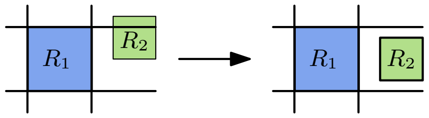

Fig. 18. 25% of the area of rectangle R_2 was in the NorthEast section of rectangle R_1. In the next layout 100% of the area of R_2 is in the East section of R_1.

We then calculate the overall relative-position-change by averaging the relative-position-change for all pairs of rectangles R_i and R_j that are in both layouts:

$$D^{rel}(L, L') = \frac{1}{|\mathcal{R}|^2} \sum_i \sum_j D^{rel}_{ij}(L, L') \qquad (2)$$

By definition, the relative-position-change is a value between 0 and 1, where 0 indicates no relevant change in the relative positions and 1 indicates the highest possible change of relative positions. Figure 19 and Figure 20 show two examples: In Figure 19 the relative positions between rectangles do not change, and thus the relative-position-change is 0 (in comparison, the layout-distance-change is 5.325). In Figure 20 the relative positions change significantly and the relative-position-change is 0.45 (in comparison, the layout-distance-change is nearly the

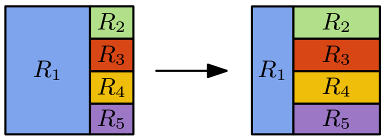

Fig. 19. A change occurred in the layout, but all relative positions stay the same. It is easy to track the movement of the rectangles. The relative-position-change is 0 and the layout-distance-change is 5.325.

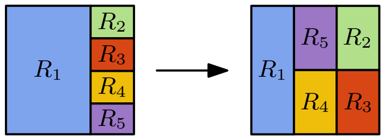

Fig. 20. A change occurred in the layout which affects the relative positions. It now becomes more difficult to track the movement of the rectangles. The relative-position-change is 0.45 and the layout-distance-change is 5.308.

same as for Fig. 19, namely 5.308). Note that it is much easier to keep track of the rectangles in Figure 19 than in Figure 20, which is clearly reflected in our stability measure.

Independently, Hahn et al. [9] very recently proposed *relative-direction-change* as a measure for stability. Relative-direction-change measures the difference in angles between the centroids of two rectangles R_i and R_j in layouts L and L'. Both this measure and our relative-position-change capture the change in relative positions, but there are two major differences. First, our measure has a cutoff point for measuring the change in relative position: we do not distinguish between R_j moving from North to East with respect to R_i or from North to South. We believe that in the former case it is already very difficult to recover the position of R_j from R_i, and not significantly less difficult than in the latter case. The second major difference is that our measure is focused purely on rectangular treemaps, whereas the relative direction change measure is defined for general treemaps. As a result, our measure is more suitable for rectangles. Note that, when considering rectangles, the exact angle between the centroids of two rectangles is not that relevant for the perception of relative position. Instead, it is more readily perceived if one rectangle is above/below or left/right from another rectangle. This aspect is covered much more accurately by our measure.

7 Experimental evaluation

We evaluate the visual quality and the stability of our incremental algorithm by comparing it to existing rectangular treemapping algorithms. More specifically, we compare the algorithms in terms of the average median aspect ratio, the average mean aspect ratio, the average relative-stability-change and the average layout-distance-change computed over an entire dataset. For this evaluation we use two different real-world datasets: the Coffee dataset and the Names dataset. The complete source code of our implementation can be found here `https://gitaga.win.tue.nl/max/IncrementalTreemap`.

Coffee dataset. The first dataset consists of the amount of coffee a country imported in the period 1994-2014 and originates from the UN comtrade database [17]. It contains all the 86 countries which have complete data in this period and has a 3-level hierarchy consisting of the country, the region it belongs to, and the continent it belongs to.

For each algorithm the average relative-position-change and the average layout-distance-change on the Coffee dataset are shown in Figure 21. While there are differences between the distribution of the two stability measures, there are no large discrepancies. It thus seems that the average relative-position-change captures similar trends of stability as the layout-distance-change if the scores are high. On low scores, the differences become more apparent as the relative-position-change score is indeed far closer to 0 for the slice-and-dice algorithm than the layout-distance-change score. This is due to the fact that the slice-and-dice algorithm does not change order-equivalence classes at all if there is no hierarchy, and only up to a limited degree if there is a hierarchy. From Figure 21 it moreover follows that the incremental algorithm significantly outperforms all other algorithms, except for slice-and-dice, on both the average relative-position-change and the average layout-distance-change.

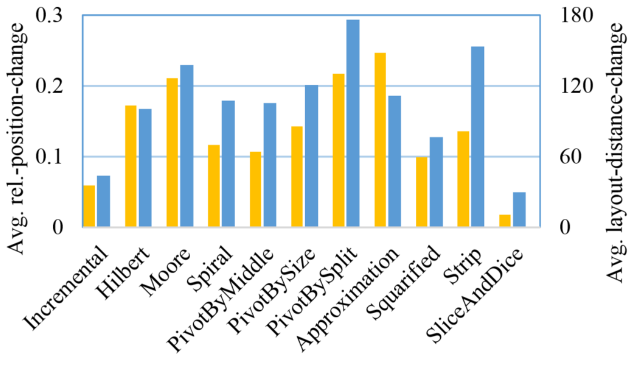

Fig. 21. A comparison between the average relative-position-change and the average layout-distance-change of the different algorithms on the Coffee dataset. The average relative-position-change is depicted by the left columns and the average layout-distance-change is depicted by the right columns. The data is averaged over the treemaps generated for each year in the Coffee dataset.

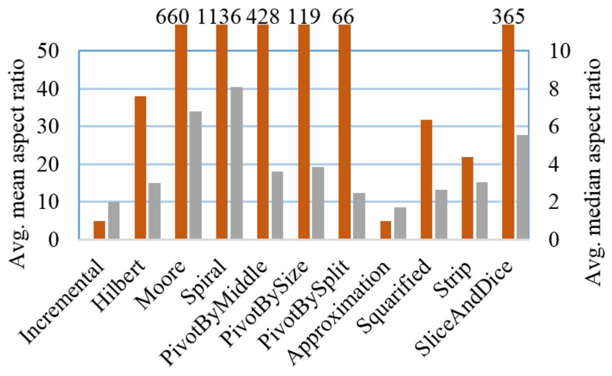

Fig. 22. A comparison between the average mean and average median aspect ratio of the different algorithms on the Coffee dataset. The average mean aspect ratio is depicted by the left columns and average median aspect ratio is depicted by the right columns. The data is averaged over the treemaps generated for each year in the Coffee dataset.

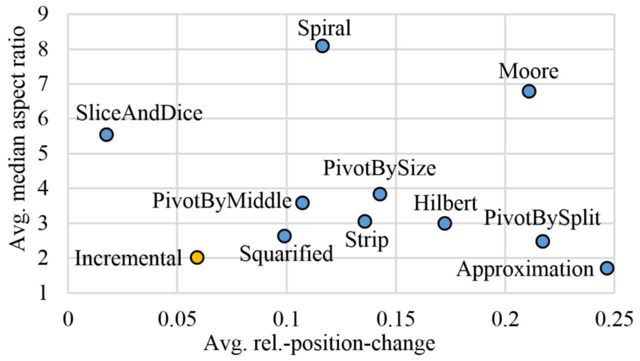

Fig. 23. A comparison between the average median aspect ratio and the average relative-position-change on the Coffee dataset. The data is averaged over the treemaps generated for each year in the Coffee dataset.

In Figure 22 we show the difference between the average mean aspect ratio and the average median aspect ratio for different algorithms. Note that for most algorithms the mean aspect ratio is significantly larger than the median aspect ratio. This implies that for each of these algorithms there are a number of rectangles with very large aspect ratios. Thus, these algorithms do not keep the maximum aspect ratio low, which results in treemaps with low visual quality. In contrast, the incremental algorithm and the approximation algorithm both do aim to minimize the maximum aspect ratio. As a result, all rectangles in the corresponding treemaps have relatively lower aspect ratios and are better visible, leading to treemaps with high visual quality.

Finally in Figure 23 the average median aspect ratio is compared to the average relative-position-change. From this figure we can see that the incremental algorithm outperforms all other algorithms on either the average median aspect ratio or average relative-position-change, and actually outperforms most algorithms on both fronts.

Names dataset. The second dataset consists of the 200 most popular boys and girls baby names in the Netherlands for each year in the period 1993-2015 and originates from the Nederlandse Voornamenbank [11].

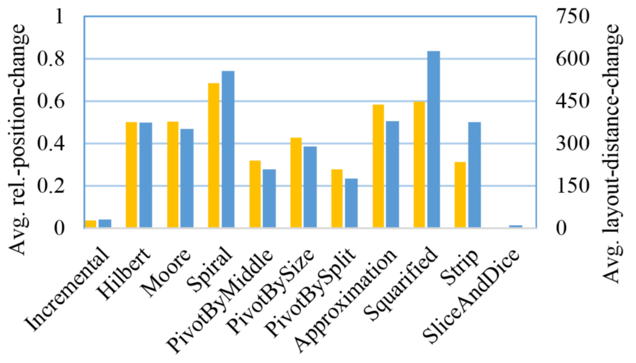

Fig. 24. A comparison between the average relative-position-change and the average layout-distance-change of the different algorithms on the Names dataset. The average relative-position-change is depicted by the left columns and the average layout-distance-change is depicted by the right columns. The data is averaged over the treemaps generated for each year in the Names dataset.

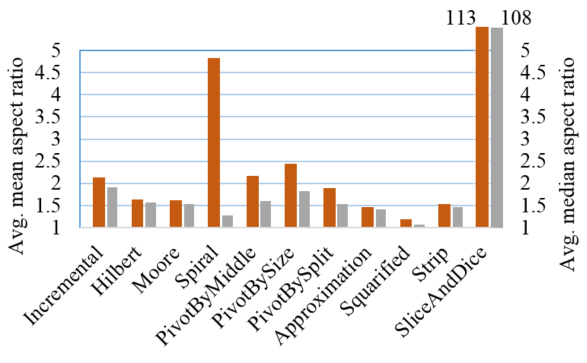

Fig. 25. A comparison between the average mean and average median aspect ratio of the different algorithms on the trimmed Names dataset. The trimmed dataset only contains the names which occur in every year of the dataset. The average mean aspect ratio is depicted by the left columns and average median aspect ratio is depicted by the right columns. The data is averaged over the treemaps generated for each year in the Names dataset.

The dataset in total contains 372 names and does not have a hierarchy. See Figure 27 for some treemaps computed with our incremental algorithm, and the Hilbert and squarified treemapping algorithms.[3]

For each algorithm the average relative-position-change and the average layout-distance-change on the Names dataset are shown in Figure 24. The discrepancies between the two measures are less pronounced in the Names dataset than in the Coffee dataset. This is caused by the fact that the Names dataset has many rectangles of similar sizes. If one large rectangle changes position, then this generally causes a large layout-distance-change and a relatively low relative-position-change, whereas with rectangles of similar sizes, this difference is much smaller. From Figure 24 we moreover see that the incremental and slice and dice algorithms are almost completely stable. This is because the Names dataset itself is quite stable. The most frequent changes that occur are insertions and deletions from names in the dataset. As the incremental algorithm uses the previous layout as a basis, the layout stays virtually the same after an insertion or a deletion. For the slice and dice algorithm the layout also stays roughly the same, as it only uses a different layout when the dataset has a hierarchy. For all the other algorithms, however, the layout can change drastically when an element is inserted or deleted, which is reflected in the stability measures.

In Figure 25 we show the difference between the average mean aspect ratio and the average median aspect ratio for different algorithms. The discrepancies between the two are quite low with the notable exception of the Spiral algorithm. Moreover the values of the mean and median aspect ratios are very low as well, which indicates that the resulting rectangles in the treemap for all algorithms except slice and dice have low aspect ratios. The incremental algorithm performs slightly worse on the aspect ratio than most other algorithms. This is mostly due to the fact that we allowed the incremental algorithm to perform only 4 local moves per timestep, while the number of rectangles per level (there is only a single level) is far larger. It is thus not always possible to immediately improve the aspect ratios of all the bad rectangles. By allowing more moves, the aspect ratio would decrease, but the stability measure would increase. The algorithm thus clearly has a tradeoff between the stability and the visual quality. In contrast, the other algorithms can change the entire layout and thus insertions and deletions do not have a similar impact on the mean and median aspect ratios.

Finally in Figure 26 the average median aspect ratio is compared to the average relative-position-change. From this figure we can see that the incremental algorithm again outperforms all other algorithms on either the median aspect ratio or the average relative-position-change. Moreover, the average relative-position-change is significantly lower than most algorithms while the average median aspect ratio is only slightly higher than most other algorithms.

The experiments show that our incremental algorithm performs very well on real-world data with respect to stability and visual quality. The algorithm obtains an average median aspect ratio below 2 for both datasets, which indicates that the rectangles have a high visual quality. Moreover, the incremental algorithm outperforms all algorithms, except for slice and dice, on both the average layout-distance-change and the average relative-position-change on both datasets. From Figure 23 and Figure 26 we can further conclude that the incremental algorithm outperforms existing treemapping algorithms on either the visual quality and/or stability in practice. This is also clearly demonstrated in the supplementary video.

The only main disadvantage of the incremental algorithm is the running time compared to the existing rectangular treemapping algorithms. As the incremental algorithm explores various possible layouts, it is slower than other rectangular treemapping algorithms that directly construct a layout. This difference in running time can become noticeable once the number of rectangles in a treemap becomes large. When a treemap contains 200 rectangles, existing treemap algorithms can generate the layout almost instantaneously, whereas the incremental algorithm takes about 15 seconds to generate the layout in our implementation. If the application is very time-sensitive, then this might become a problem. In that case the incremental algorithm can be sped up by changing the parameters of the exploration, but the quality of the

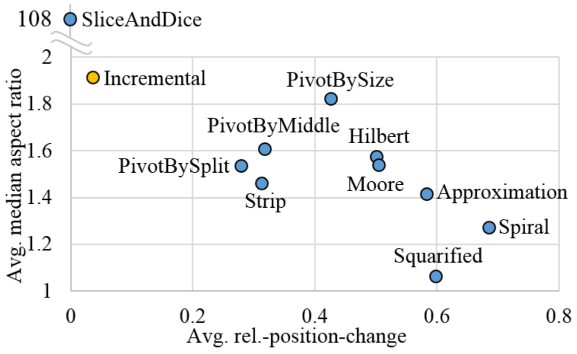

Fig. 26. A comparison between the average median aspect ratio and the average relative-position-change. on the Names dataset. The data is averaged over the treemaps generated for each year in the Names dataset.

[3]In Figure 27 the dataset is trimmed to only those names that occur in every year, such that there are no insertions or deletions that influence the stability.

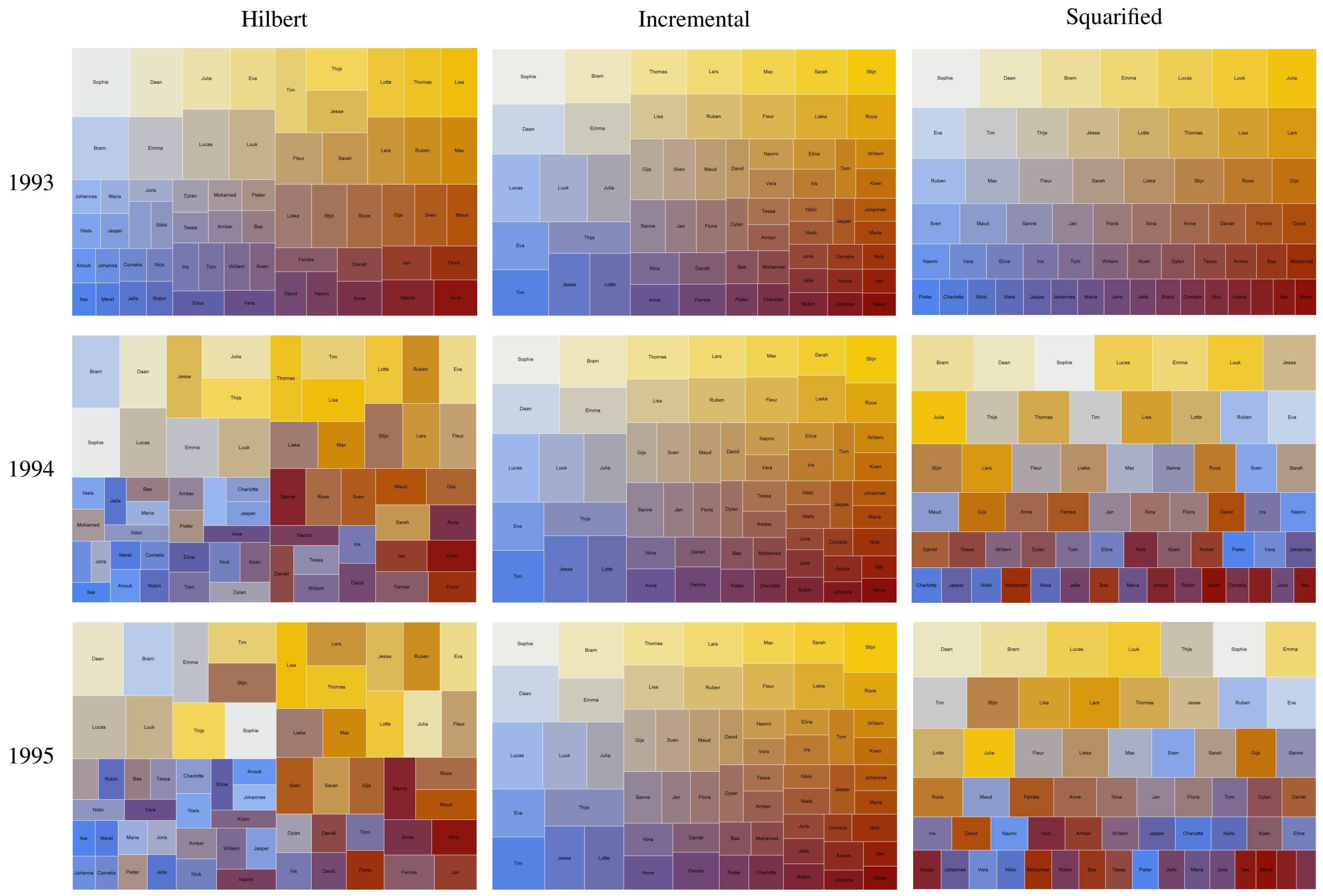

Fig. 27. A comparison of the Hilbert treemapping algorithm, our incremental treemapping algorithm, and the squarified treemapping algorithm on the trimmed Names dataset for the years 1993-1995. The dataset is trimmed to only those names that occur in every year, such that no insertions or deletions occur. One can observe that our incremental algorithm is significantly more stable than the other algorithms. This difference in stability is even more apparent in the supplementary video.

resulting treemaps would deteriorate accordingly. However, note that with a large number of rectangles in a treemap, the treemap becomes increasingly hard to read. In most reasonable cases, the incremental treemapping algorithm will generally be sufficiently fast, even for interactive treemap generation. For example, for the Coffee dataset it takes only 62 ms on average to compute the treemap for one time step.

8 Conclusion

We presented a new algorithm to compute stable treemaps. Our algorithm is based on the concept of local moves, modifications to the treemap that influence only a small part of the treemap. These local moves allow us, in contrast to existing treemapping algorithms, to explore the full range of options for choosing layouts, which can provably lead to treemaps with better visual quality. Furthermore, the local moves allow us to control the tradeoff between stability and visual quality, simply by limiting the number of local moves between every two time steps. Our experiments show that our incremental algorithm does not only perform better with respect to stability and/or visual quality in theory, but also in practice on real-world data. The only main disadvantage of our algorithm is its running time, which may be prohibitive for interactive applications on very large datasets. Nonetheless, for most reasonable practical scenarios, where the treemaps still need to remain readable, our algorithm is sufficiently fast. Beyond that, it is even possible to control the tradeoff between the running time of the algorithm and the visual quality of the treemaps, again by controlling the number of local moves between every two time steps. A fully controllable tradeoff between the three aspects visual quality, stability, and running time remains an interesting open problem.

Acknowledgments

The Netherlands Organisation for Scientific Research (NWO) is supporting M. Sondag and B. Speckmann under project no. 639.023.208, and K. Verbeek under project no. 639.021.541.

References

[1] M. Balzer, O. Deussen, and C. Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proceedings of the ACM Symposium on Software Visualization*, pages 165–172, 2005.

[2] 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.

[3] M. Bruls, K. Huizing, and J. J. Van Wijk. Squarified treemaps. In *Data Visualization*, pages 33–42. Springer, 2000.

[4] F. R. Chung, R. L. Graham, V. Hoggatt, and M. Kleiman. The number of Baxter permutations. *Journal of Combinatorial Theory, Series A*, 24(3):382–394, 1978.

[5] M. De Berg, B. Speckmann, and V. Van Der Weele. Treemaps with bounded aspect ratio. *Computational Geometry*, 47(6):683–693, 2014.

[6] 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*, 2005.

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

[8] A. Erdélyi and I. M. Etherington. Some problems of non-associative combinations (2). *Edinburgh Mathematical Notes*, 32:7–14, 1940.

[9] S. Hahn, J. Bethge, and J. Döllner. Relative direction change: A topology-based metric for layout stability in treemaps. In *Proceedings of the 8th International Conference of Information Visualization Theory and Applications (IVAPP 2017)*, pages 88–95, 2017.

[10] S. Hahn, J. Trümper, D. Moritz, and J. Döllner. Visualization of varying hierarchies by stable layout of voronoi treemaps. In *International Conference on Information Visualization Theory and Applications*, pages 50–58. IEEE, 2014.

[11] Meertens Instituut, KNAW. Nederlandse voornamenbank. `https://www.meertens.knaw.nl/nvb`, 2013. Accessed on 30-05-2016.

[12] 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.

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

[14] B. Shneiderman and M. Wattenberg. Ordered treemap layouts. In *Proceedings of the IEEE Symposium on Information Visualization*, page 73, 2001.

[15] S. Tak and A. Cockburn. Enhanced spatial stability with Hilbert and Moore treemaps. *IEEE Transactions on Visualization and Computer Graphics*, 19(1):141–148, 2013.

[16] 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.

[17] United Nations. UN comtrade database. `https://comtrade.un.org`, 2016. Accessed on 15-02-2017.

[18] M. Wattenberg. A note on space-filling visualizations and space-filling curves. In *Symposium on Information Visualization*, pages 181–186. IEEE, 2005.

[19] B. Yao, H. Chen, C.-K. Cheng, and R. Graham. Floorplan representations: Complexity and connections. *ACM Transactions on Design Automation of Electronic Systems*, 8(1):55–80, 2003.

[20] E. F. Young, C. C. Chu, and Z. C. Shen. Twin binary sequences: A nonredundant representation for general nonslicing floorplan. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(4):457–469, 2003.