MASTER

Stability of treemap algorithms

Sondag, M.F.M.

Award date:
2016

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
Stability of treemap algorithms

Master Thesis

by

M.F.M. Sondag

Committee:
prof.dr. B. Speckmann
dr. K.A.B. Verbeek
dr. M.A. Westenberg

Supervisors:
prof.dr. B. Speckmann
dr. K.A.B. Verbeek

Eindhoven, Tuesday 4th October, 2016
To evaluate the quality of a treemap algorithm, two quality metrics should be considered. The first quality metric is the aspect ratio of the treemaps generated by the algorithm. The aspect ratio determines how accurate the areas of the rectangles in the treemap can be interpreted by the user. The second quality metric, which is the main focus of this thesis, is the stability score. The stability score indicates how hard it is to track rectangles when the treemap has changed. The aim of this thesis is twofold. The first aim is to evaluate how the stability score can be objectively determined. The second aim is to develop a treemap algorithm that is able to generate treemap that have both low stability scores and low aspect ratios.

To objectively determine stability score we first evaluated the existing definitions of stability. We noticed that none of the current definitions take the structures in the treemap into account. We then developed a new definition for the stability score, that does take these structure into account. The new definition is based on the change in the relative positions of the rectangles with regard to each other.

To develop new treemap algorithms that are able to generate treemaps that have both a low stability score and a good aspect ratio, we introduced the concept of local moves. Local moves manipulate an existing layout in such a way that the resulting layout only differs slightly compared to the original layout. Using a sequence of local moves, it is moreover possible to manipulate the layout such that it is possible to reach any layout from any layout. We developed two new treemap algorithms using these local moves. The newly developed algorithm are the first treemap algorithm that can generate non-sliceable treemaps. In terms of the stability score, the newly developed algorithm outperform all current practical treemap algorithms on both artificial and real world dataset. Moreover the newly developed algorithms perform well on the aspect ratio quality measure as well.
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.1 Related literature</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1.2 Preliminaries</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>1.2.1 Rectangular treemaps</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>1.2.2 Types of rectangular treemaps</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>1.2.3 Twin binary sequence</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1.2.4 Order equivalence graph</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>1.3 Results and organization</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>Stability</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>2.1 Contribution factors of stability</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>2.2 Current definitions</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>2.2.1 Layout distance change function</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>2.2.2 Variants of the layout distance change function</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>2.3 A new definition of stability</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>Stable algorithm</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>3.1 Local Moves</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>3.1.1 Types of local moves</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>3.1.2 All rectangular layouts are reachable using local moves</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>3.2 Incremental moves algorithm</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>3.2.1 Approximation algorithm</td>
<td>33</td>
</tr>
</tbody>
</table>

Stability of treemap algorithms
CONTENTS

A.1.2 non-sliceable rectangular layouts ........................................... 86

B Lower bounds on minimal maximum aspect ratio of rectangular layouts 89
  B.1 Aspect ratios of sliceable and non-sliceable rectangular layouts ........ 89
    B.1.1 Lower bound on the aspect ratio of sliceable rectangular layouts . 89
    B.1.2 The lower bound is tight .............................................. 93
    B.1.3 non-sliceable layouts can outperform optimal sliceable layouts .... 94

Stability of treemap algorithms
Chapter 1

Introduction

As data becomes more readily available it is important to be able to extract the information contained in the data. One common way to analyze the data is by making a visualization of it to give insight into the data in an intuitive way. One of the most common types of visualization is the pie chart. The pie chart is mostly used to display the relative distribution of the items. An example of a pie chart is shown in Figure 1.1. In Figure 1.1 we see the distributions of the audience ratings of the top 10 Dutch tv channels in 2005 [19]. Pie charts are reasonable visualizations for comparing percentages of a small number of items to each other. However, when the number of items that need to be compared increases it becomes ever more difficult to distinguish the different slices from each other, and it becomes nearly impossible to determine the areas of the slices. If we want to split the data for each tv channel further into the audience ratings of the individual programs of the channels, a large number of additional items need to be visualized. To visualize a large number of items we can use a treemap.

A treemap is a visualization method where a rectangle is partitioned into a number of sub-

![Figure 1.1: A pie chart displaying the distribution of audience ratings of the top 10 Dutch tv channels in 2005.](image)
CHAPTER 1. INTRODUCTION

rectangles. Each subrectangle represents an item in the data. The area of a subrectangle represents the value of the item. Additionally, each subrectangle can again be recursively partitioned to visualize hierarchical data. In this thesis we will however restrict ourselves to single-level treemaps, that is treemaps without hierarchical data.

The quality of a treemap can be determined using the maximum aspect ratio of the rectangles in the treemap. Extreme aspect ratios should be avoided as the accuracy of determining the area of a rectangle is decreased when the aspect ratio is high as was shown by Kong, Heer and Agrawala [12]. The maximal aspect ratio should thus be kept low such that the areas can be accurately compared.

An example of a treemap visualization where the maximal aspect ratio is low is shown in Figure 1.2. The areas in this treemap encode the audience ratings of the top 10 Dutch tv channels in 2005. As all the aspect ratios are low, it is not too hard to determine the relative sizes of the areas.

When the aspect ratios become larger, as is shown in Figure 1.3, it becomes hard to determine the relative sizes of the areas. For example, we look at the rectangles "Ned3" and "RTL7" in Figure 1.3. It is almost impossible to tell which one of the two rectangles is larger in Figure 1.3 while in Figure 1.2 this was not a problem.

A large amount of data available is furthermore time dependent. Therefore, it is often required to be able to analyze the changes in the data over time as well. One can use treemaps for this kind of analysis as well by generating a treemap for each timestep. The analysis can then be performed by comparing the treemaps with each other.

To be able to compare two treemaps we ideally want the structure of the treemaps to be roughly the same. If the structure is roughly equal then we can more easily find a rectangle in both treemaps as it is in the same place in both treemaps. To determine how hard it is to keep track of rectangles from the previous treemap to the current treemap a second quality metric is required which we will denote as the stability score. The lower the stability score, the easier it is to keep track of the rectangles and the more stable the

Figure 1.2: A treemap with low aspect ratios displaying the distribution of audience ratings of the top 10 Dutch tv channels in 2005.

Stability of treemap algorithms
Figure 1.3: A treemap with a high aspect ratios displaying the distribution of audience ratings of the top 10 Dutch tv channels in 2005.

treemap is. Ideally, we would want to be able to easily keep track of all the rectangles such that the analysis becomes easier. The stability score should thus be as low as possible.

An example of the difference between a treemap with a high stability score and a low stability score is shown in Figure 1.4. In Figure 1.4 the audience ratings of the top 10 Dutch tv channels in 2005 and 2015 are visualized. In Figure 1.4b the structure of the treemap has changed while in Figure 1.4c the structure of the treemap is mostly maintained. It is a lot harder to keep track of the rectangles from Figure 1.4a towards Figure 1.4b than it is from Figure 1.4a towards Figure 1.4c.

To determine the stability score, a number of measures currently exist. However, none of these measures covers all the aspects of stability as is described in detail in Section 2. The most crucial aspect that the current definitions are missing is that they only consider rectangles individually. They do not account for the change in the structure of groups of rectangles in the treemap. In Section 2 we will develop a new stability measure that takes this change of structure into account.

All current treemap algorithms generate a completely new treemap for each step. It however seems to make sense to modify the previous treemap to improve the aspect ratio while maintaining the stability score, instead of generating an entirely new treemap. By using the previous treemap as a basis for the new treemap one can control how stable the resulting treemap is by controlling how much one modifies the treemap. It thus allows one to balance the stability and the aspect ratio in a controlled way. A method to modify an existing treemap and two treemap algorithms which use this method are presented in Section 3.
Figure 1.4: A treemap displaying the distribution of the top 10 Dutch tv channels of 2005, and a stable and an unstable treemap displaying the distribution for 2015.
1.1 Related literature

Treemap algorithms have been around since 1991 after the introduction of them by Schneiderman in the form of the Slice and Dice algorithm [17]. The Slice and Dice algorithm however suffered from rectangles with a high aspect ratio. In response to this problem the Squarified treemap [5] was developed by Bruls, Huizing and Van Wijk which minimizes the aspect ratio using a heuristic approach. To give guarantees on the aspect ratio the Approximation algorithm [15] was developed by Nagamochi and Abe. Finally De Berg, Speckmann and Van Der Weele proved that obtaining the optimal maximum aspect ratio for rectangular treemaps is strongly NP-complete [7].

After the development of the Squarified treemap algorithm, which is able to generate treemaps with low aspect ratios, the research shifted. The focus was now not only on the aspect ratio, but also on the stability of the treemaps.

A first definition for the stability score was given by Schneiderman and Wattenberg which is the layout distance change function score [18]. The layout distance change function score measures how much each rectangles moves. A number of complementary functions to determine the stability score were developed as well. The variance distance change function [20] which measures the variance of the changes in distance was developed by Tak and Cockburn. A second variant that Tak and Cockburn introduced was the locational drift measure [20] which measures the stability over a larger period of time. Additionally a definition for the stability score for non-rectangular treemaps was proposed by Hahn, et al. which measures the distance using the centroids of shapes as the basis [11].

Schneiderman and Wattenberg also introduced the ordered treemaps algorithms [18]. In an ordered treemap, rectangles that are near each other in the input are placed near each other in the treemap. This reduces the instability over time.

A number of ordered treemaps were then developed, namely the Pivot-by-(Middle, Size and Split-Size) algorithms [18] by Schneiderman and Wattenberg, the Strip algorithm [2] by Bederson, Schneiderman and Wattenberg, the Split algorithm [8] by Engdahl, the Spiral algorithm [21] by Tu and Shen and the Hilbert and Moore algorithms [20] by Tak and Cockburn.

To measure the success of maintaining the underlying order, Bederson, et al. introduced the readability metric [2]. The readability metric measures how often the motion of the readers eye changes direction as the treemap is scanned in order. Tu and Shen additionally introduced the continuity metric [21] 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 a layout to find a particular item in an ordered treemap.

Generating non-rectangular treemaps instead of rectangular treemaps was researched as well. Example of non-rectangular treemaps are the voronoi treemaps [1] as presented by Balzer, Deussen en Lewerentz, the orthoconvex and L-shaped treemaps [7] by De Berg, et al. and the Jigsaw treemap [22] as presented by Wattenberg.
Visual enhancements of treemaps have been researched as well. Examples of these are the 3-dimensional treemaps [4] by Bladh, Carr and Scholl, the animation of 3-dimensional treemaps [3] by Bladh, Carr and Kljun, the Cascaded treemaps [13] by Lu and Fogarty and finally the Cushion treemaps [5] by Bruls et al.

An independent line of research focused to find the number of equivalence classes of layouts for a given number of rectangles and on methods to enumerate these equivalence classes. Two layouts are equivalent if each rectangle in both layouts has the same adjacencies with the other rectangles in the layout.

Yao, et al. described the twin binary tree sequence [23] which can be used to represent an equivalence class of a layout. Young, Chu and Shen further expanded on the twin binary tree sequence by presenting a method to transform any layout to any other layout given that the labeling of the rectangles does not matter [24]. The twin binary sequence structure is explained in detail in Section 1.2.3. Using the twin binary tree sequence the exact number of sliceable and non-sliceable treemaps were found by Yao et al. [23]. The amount of sliceable treemaps equals the Baxter number [6] as presented by Chung, et al. The amount of non-sliceable treemaps equals the Schröder number [10] as presented by Erdélyi and Etherington. Sliceable and non-sliceable treemaps are explained in detail in Section 1.2.2.

1.2 Preliminaries

1.2.1 Rectangular treemaps

To define the treemapping problem we will let $R_0$ denote the input rectangle and we let $\mathcal{R}$ denote the set of input rectangles where each rectangle $r \in \mathcal{R}$ has a non negative size $s(r)$.

Without loss of generality we will assume that the sizes of the rectangle are normalized, that is:

$$\sum_{r \in \mathcal{R}} s(r) = \text{area}(R_0)$$

The treemapping problem can then formally be defined as follows:

**Definition 1.** The treemapping problem takes as input the input rectangle $R_0$, a set of rectangle $\mathcal{R}$ where each rectangle $r \in \mathcal{R}$ has a non negative size $s(r)$.

As an output it produces a layout $L$, which is a partitioning of $R_0$ using the rectangles in $\mathcal{R}$. It should hold that in the layout $L$ each rectangle $r \in \mathcal{R}$ is positioned such that $\text{width}(r) \times \text{height}(r) = s(r)$.

We denote the left x-coordinate of a rectangle $r$ as $x(r)$, the top y-coordinate as $y(r)$, the...
width as $w(r)$, the height as $h(r)$, the $(x, y)$ position of the center of the rectangle as $c(r)$, the area as $area(r)$ and the aspect ratio as $a(r) = \min \left( \frac{w(r)}{h(r)}, \frac{h(r)}{w(r)} \right)$. Furthermore we will let $r_i$ uniquely denote a rectangle in $\mathcal{R}$ for all $1 \leq i \leq |\mathcal{R}|$. Finally we let $a_{\text{max}}(L)$ denote the maximum aspect ratio in the layout.

To be able to handle changes in the data over time, all functions will additionally have a dependency on $t$. Finally the stability score between the layout generated for time $t$ and the layout generated for time $t+1$ can then be defined as $S(L(t), L(t+1))$.

### 1.2.2 Types of rectangular treemaps

Rectangular treemaps can be divided into two types of treemaps based on the maximal line segments present in the layout $L$ of the treemap. A line segment is formed by consecutive edges of the rectangles in the layout $L$. A segment is maximal if it is not contained in any other line segment. An example of such a maximal segment is shown in Figure 1.5.

![Figure 1.5: The red and blue line segments together form a maximal segment.](image-url)

We denote the set of all maximal segments in a layout $L$ as $\mathcal{MS}(L)$. We furthermore let $ms_i \in \mathcal{MS}(L)$ uniquely denote a maximal segment for all $1 \leq i \leq |\mathcal{MS}(L)|$. If a maximal segment $ms$ is part of the input rectangle $R_0$ we will denote it as a boundary maximal segment. If $ms$ is not part of the input rectangle $R_0$ we will denote it as an inner maximal segment.

We identify one degenerate case which occurs when two maximal segments intersect each other. When this occurs, we split one of the two maximal segments into two separate maximal segments that meet at the intersection point as is shown in Figure 1.6. For the remainder of this thesis we will assume that every degenerate case has been handled in this way.

We furthermore observe that a rectangular layout with $n$ rectangular regions has $n + 3$ maximal segments. Moreover, 4 of these maximal segments are boundary maximal segments and $n - 1$ maximal segments are inner maximal segments when there are no degenerate cases.

Finally, we define a rectangle $r$ to be left adjacent to a maximal segment $ms$ if the left edge of $r$ is part of $ms$, top adjacent if the top edge of $r$ is part of $ms$, right adjacent is the right edge is part of $ms$ and bottom adjacent if the bottom edge is part of $ms$. 

Stability of treemap algorithms
CHAPTER 1. INTRODUCTION

Figure 1.6: The red and blue maximal segments intersect each other which results in a degenerate case. This is handled by splitting the red maximal segment into two separate maximal segments.

**Sliceable rectangular treemaps**

Sliceable treemaps have a layout where we can recursively slice the treemap into two parts by slicing over a single maximal segment at a time. An example of a sliceable treemap and how it is sliced is shown in Figure 1.7.

As a consequence of a layout being sliceable it holds that all the rectangles in the treemap must be grounded rectangles. A rectangle $r$ in a layout $L$ is ground if at least one of its sides is a maximal segment. That is, there exists a maximal segment $ms$ such that $r$ is the only rectangle adjacent to one side of this maximal segment. The existence of such a maximal segment is proven in Appendix A.

For sliceable rectangular treemaps we can give a tight lower bound on the maximum aspect ratio, which equals $\sqrt{\frac{s(B)}{s(A)}}$ where $A$ is the rectangle with the smallest size and $B$ is the rectangle with the second smallest size as is proven in Appendix B.

**Non-sliceable rectangular treemaps**

In contrast to sliceable rectangular treemaps, non-sliceable treemaps can not always be sliced into two parts over a single maximal segment. An example of such a treemap is given in Figure 1.8.

For non-sliceable treemaps it also does not need to hold that all the rectangles in the treemap are grounded. In Figure 1.8 rectangle E is not grounded. However, it is possible
1.2. PRELIMINARIES

Figure 1.8: An example of an non-sliceable treemap. There exists no maximal segment which slices the treemap into two parts.

Figure 1.9: An example of a non-sliceable treemap consisting of only grounded rectangles.

that in an non-sliceable treemap all rectangles are grounded as is shown in Figure 1.9. As non-sliceable treemaps do not have to consist of only grounded rectangles, the lower bound on the maximum aspect ratio of $\sqrt{\frac{s(B)}{s(A)}}$ for sliceable treemaps does not hold. As is shown in Appendix B, non-sliceable treemaps can obtain maximal aspect ratios significantly smaller than this lower bound.

1.2.3 Twin binary sequence

Twin binary sequences [23] can be used to represent treemaps as shown by Yao, et al. We first explain what a twin binary tree is and afterwards we will explain how the twin binary sequences relates to the twin binary tree.

Two binary trees $t_1$ and $t_2$ are twin binary if and only if they contain the same set of nodes, and it holds that the labeling $\Theta(t_1)$ equals the inverse of the labeling $\Theta(t_2)$. The labeling $\Theta(t)$ of a binary tree $t$ defines as follows:

Initially let $\Theta(t)$ equal the empty sequence. We visit the nodes of the tree $t$ using an inorder traversal and whenever we visit a node with no left child, we add a 0 bit to $\Theta(t)$. Whenever we visit a node with no right child, we add a 1 bit to $\Theta(t)$. Finally the first 0 and the last 1 will be omitted.

A twin binary sequence is compromised of 4 parts: $tbs = (\pi, \alpha, \beta, \beta')$ which can be mapped one-to-one to a pair of twin binary trees $(t_1, t_2)$ as is shown by Young et al. [24]. The mapping from the twin binary sequence $tbs = (\pi, \alpha, \beta, \beta')$ to the pair of twin binary trees $(t_1, t_2)$ is as follows: $\pi$ equals the labels of the nodes of the twin binary tree when the nodes are visited in-order. $\alpha$ equals the labeling $\Theta(t_1)$ of $t_1$. $\beta$ is a sequence of $n$ bits, where the $i$'th bit is 0 if the node with label $\pi_i$ is a left child in $t_1$ or if it is the root of $t_1$, and 1 if it is a right child in $t_1$. $\beta'$ is a sequence of $n$ bits, where the $i$'th bit is 0 if the node
CHAPTER 1. INTRODUCTION

A pair of twin binary trees \((t_1, t_2)\) and their associated twin binary sequence \(tbs = (\pi, \alpha, \beta, \beta')\).

Figure 1.10: A layout \(L\) and the corresponding pair of twin binary trees and the corresponding twin binary sequence.

with label \(\pi_i\) if a left child in \(t_2\) or if it is the root of \(t_1\), and 1 if it is a right child in \(t_2\).

A pair of twin binary trees \((t_1, t_2)\) on its turn has a one-to-one mapping to an equivalence class of a layout \(L\) as is shown by Yao et al. [23] This mapping is as follows: If the left-top corner of a rectangle \(x\) is adjacent to the right-top corner of a rectangle \(y\), then \(y\) is a left child of \(x\) in \(t_1\). If the right-bottom corner of a rectangle \(x\) is adjacent to the right-top corner of a rectangle \(y\), then \(y\) is a right child of \(x\) in \(t_1\). If the left-top corner of a rectangle \(x\) is adjacent to the left-bottom corner of a rectangle \(y\), then \(y\) is a left child of \(y\) in \(t_2\). If the right-bottom corner of a rectangle \(x\) is adjacent to the left-bottom corner of a rectangle \(y\), then \(y\) is a right child of \(y\) in \(t_2\).

An example of a twin binary sequence, its related pair of twin binary trees, and its related layout is shown in Figure 1.10.

As a twin binary sequence \(tbs\) has a one-to-one mapping with an equivalence class of a layout \(L\), it is possible to construct a layout \(L\) from a given twin binary sequence \(tbs\) as is shown by Young et al. [24]

1.2.4 Order equivalence graph

In order to develop a stable treemap algorithm we will use the notion of order equivalence layouts. The notion of order equivalence were introduced by Eppstein, et al. [9] To determine whether two layouts \(L, L'\) are order equivalent we compare their order equivalence graph. The order equivalence graph is defined as follows:
Definition 2. The order equivalence graph $OEG(L)$ of a layout $L$ is a directed acyclic multigraph that has a vertex per maximal segment $ms$, an edge from the segment on the left boundary of each rectangle $r \in \mathcal{R}$ to the segment on the right boundary of the same rectangle $r$, and an edge from the segment on the bottom boundary of each rectangle $r \in \mathcal{R}$ to the segment on the top boundary of the same rectangle $r$.

An order equivalence graph thus consists of a partial order on the vertical maximal segments and a partial order on the horizontal maximal segments. An example of two layouts that are not equivalent to each other, but which are order equivalent to each other is shown in Figure 1.11.

![Figure 1.11: Two non-identical layouts $L$ and $L'$ which have the same order equivalence graph: $OEG(L) = OEG(L')$. The red arrows show the horizontal partial order and the blue arrows show the vertical partial order.](image)

Eppstein et al. [9] proved that given a layout $L$ and a size function $s$, we can calculate the positions of the rectangles of $L$ such that the size function $s$ is realized and the resulting treemap has a layout $L'$ that is order equivalent to $L$.

Given a layout $L$ we will determine the positions of the rectangles to realize the size function $s$ by using the hill climbing algorithm as was described by Eppstein et al. [9] The algorithm works as follows:

To correct the sizes of the rectangles without change the order equivalence graph of the layout $L$, we incrementally move the inner maximal segments in $L$. In each step we calculate how far we must move each maximal segment $ms \in MS(L)$ by solving the matrix equation $SR \ast X = Y$ for $X$.

$SR$ encodes for each inner maximal segment $ms$ how much a shift of 1 in the x/y of the position of $ms$ will change the areas of the rectangles in the graph. If $ms$ is horizontal, then the shift will be on the y-coordinate of $ms$. If $ms$ is vertical the shift will be on the x-coordinate of $ms$. The boundary maximal segments are not considered as these cannot be moved. The rectangles will be represented as rows and the maximal segments will be represented as columns resulting in a $|\mathcal{R}|$ by $|\mathcal{R}| - 1$ matrix as there are $|\mathcal{R}|$ rectangles.
and \(|R - 1|\) inner maximal segments. One additional zero column is added to the end to make the matrix a square matrix.

\(Y\) will encode for each rectangle in the layout \(L'\) the number its size is off from the size it should have according to \(s\). The rectangles will be represented as rows resulting in a \(|R|\) by 1 vector.

Finally \(X\) will encode how much we must shift the maximal segments to result in a state where all rectangles have a size difference of 0 with \(s\). As the problem is bilinear, simply shifting the maximal segments according to \(X\) will not solve the problem exactly. Shifting horizontal maximal segments will affect how much vertical maximal segments are adjacent to the rectangles and the other way around. We therefore first scale the vector \(X\) by a scalar value \(\epsilon < 1\) before shifting the maximal segments according to \(X\). After shifting the maximal segments according to \(X\) we will update the position of the rectangles adjacent to the maximal segments. For each rectangle \(r\) we will set its position to be equal to the rectangle formed by the 4 maximal segments this rectangle is adjacent to. This results in the update layout \(L'\).

After shifting the maximal segments we will check if the resulting layout is still valid and whether \(L'\) is closer to \(s\) than \(L\). A layout is valid if none of the rectangles in the layout are overlapping each other. When a maximal segment is shifted too far, it can occur that a maximal segment \(ms_1\) that should be to the left of the maximal segment \(ms_2\) is now to the right of \(ms_2\). When this happens rectangles that are left adjacent to \(ms_2\) will overlap rectangles that are right adjacent to \(ms_2\) and left adjacent to \(ms_1\) as the positions of the rectangles are determined through the positions of the maximal segments. An example of this is shown in Figure 1.12.

![Figure 1.12: ms1 is shifted too far to the right. This results in an invalid layout as both rectangles A and B are then positioned to the top-rights of ms2 which makes them overlap each other.](image)

If the resulting layout is not valid, or the sizes of the rectangles in \(L\) are closer to \(s\) than the sizes of the rectangles in \(L'\), then we have shifted the maximal segments too much. We will then retry shifting the segments with a smaller \(\epsilon\) in the next iteration. Otherwise \(L'\) is a valid layout and closer to \(L\). We will then increase \(\epsilon\) slightly to speed up the shifting and use layout \(L'\) as our new base layout. We will keep shifting the maximal segments using these steps, until the rectangles in \(L'\) all have sizes within a factor \(c\) to the size function \(s\). Alternatively it can occur that we do not have a layout \(L\), but only have an order equival-
ence graph $G$. In this case we first generate an initial layout $L$ from the order equivalence graph $G$ by transforming the order equivalence graph into a twin binary sequence $T$. From the twin binary sequence $T$ we can then generate a layout $L$ using the algorithm as presented by Young, et al. [24]

1.3 Results and organization

In this thesis we are going to consider two problems.

The first problem that we are going to consider is developing a measure that determines the stability score between two layouts. In Section 2 we will go through the current definitions of stability and show that they fail to consider structures of rectangles within the treemap. We will describe a new measure of stability that is based on the relative positions of the rectangles with regard to one other that does cover these structures.

The second problem that we are going to consider is developing a treemap generation method that uses an existing treemap as a basis instead of regenerating the entire treemap. In Section 3 we will present the concept of local moves which are moves on the treemap that change the treemap by a limited amount at a time. We will furthermore present two algorithms that use these local moves. The first algorithm uses only the local moves themselves to maintain the aspect ratios. The second algorithm will first divide the input into groups of a certain size to create a hierarchical structure. To modify the treemap it will use a combination of the approximation algorithm as presented by Nagamochi and Abe [15], and local moves on the rectangles within these groups to maintain the aspect ratios.

Both these algorithms are additionally capable of producing non-sliceable treemaps alongside with sliceable treemaps which no treemap algorithm was capable of.

To evaluate the two new treemap algorithms we will perform experiments to determine their performance in terms of aspect ratio and stability score in Section 4. From these experiments we can conclude that the new treemap algorithms achieve a very low stability score and a very low aspect ratio. Moreover, they outperform all current existing algorithms in obtaining a balance between the stability and the aspect ratio.
Chapter 2

Stability

In order to develop a new definition of stability we first look at the factors that contribute to the stability. We then evaluate the current definitions of stability and show why they are lacking as a definition of stability. Finally, we present a new definition of stability that covers the weaknesses of the existing definitions of stability.

2.1 Contribution factors of stability

The stability can be intuitively defined as how hard it is to keep track of the rectangles in the treemap. We believe that there are a number of factors that contribute to whether a layout is stable given another layout.

The first factor that contributes to the stability is the change in spatial positions of the rectangles between the current layout and the previous layout. If a rectangle jumps in spatial position, it is no longer in the position expected. The user must then search for this rectangle again. The more the spatial position of a rectangle changes, the harder it is to find this specific rectangle again.

The second factor that contributes to the stability is the change in the shapes of the rectangles. If the shape of a rectangle changes from a square to a thin and long rectangle in the new layout, it becomes harder to track this rectangle due to the visual inconsistency. Similarly, if the area of a rectangle changes from a large area to a small area, it becomes harder to track the rectangle as well.

The third and final factor that contributes to the stability is the change of group structures in the treemap. Group structures in a treemap are groups of rectangles which are positioned according to a pattern. If these patterns are maintained in the new layout it becomes easier to track the rectangles. Instead of tracking a specific rectangle $r_1$, the user now only needs to track a group structure to which the rectangle $r_1$ belongs to know the
Figure 2.1: The group structure is unchanged while the spatial position and the form have changed a lot.

general location of the rectangle \( r_1 \). As the group structures consist of several rectangles it is far easier to track the group structures than the rectangle itself. An example of this is shown in Figure 2.1. In Figure 2.1 the form and the spatial positions of the rectangles have changed quite a lot, but the group structures are still intact. Examples of these group structures in Figure 2.1 are that rectangles C,D,E,F,G are between rectangle A and B, Rectangle E and D are to the left of F, Rectangle E is above rectangle D, etc. Even though the shape and the spatial positions have changed quite a lot, it is still easy to track the rectangles in the treemap.

It thus seems that the change of the group structures is an important contributor to the stability of the layout. As long as the group structure stays intact, the impact of the change in shape and/or spatial position on the perceived stability seems to be relatively low. However, to determine exactly how much impact each factor has on the perceived stability a user experiment would be required which is outside the scope of this thesis.

### 2.2 Current definitions

There is currently one major definition of stability for rectangular treemaps, namely the layout distance change function \([2]\) as presented by Bederson et al. The layout distance change function essentially measures the average distance between each pair of corresponding rectangles in two different layouts \( L(t), L(t') \).

#### 2.2.1 Layout distance change function

The layout distance change function is defined as follows: Consider two layouts at two consecutive points in time \( L(t) \) and \( L(t') \). Let \( X \) be the set of rectangle in \( (R(t) \cap R(t')) \), i.e. the set of rectangles present in both layouts. Furthermore, let \( d(R_1, R_2) \) denote the Euclidean distance between the two rectangles \( r_1, r_2 \):

\[
d(r_1, r_2) = \sqrt{(x(r_1) - x(r_2))^2 + (y(r_1) - y(r_2))^2 + (w(r_1) - w(r_2))^2 + (h(r_1) - h(r_2))^2}
\]
2.2. CURRENT DEFINITIONS

The layout distance change function can then be written down as:

\[
LayoutDistanceChange(L(t), L(t+1)) = \frac{1}{|X|} \sum_{r \in X} d(L(t,r), L(t+1,r))
\]

As an example we take the layouts as shown in Figure 2.2. In this figure two different changes to the layout are considered. In Figure 2.2a the layout changes slightly and has a stability score of 9. In Figure 2.2b there is a large change to the layout as rectangles D and E are now vertically stacked instead of horizontally. This results in a stability score of 40 which is indeed a higher stability score.

The layout distance change score thus seems to cover the notion of stability to some degree.

Shortcomings in the definition

The layout distance change function score has a number of shortcomings which make it unable to serve as a complete definition of stability.

Figure 2.2: The score of the layout distance change function of a small and a large change in the layout.
The first major problem with the definition is that it mixes the dimensions of the area and spatial positioning incorrectly. Due to this mixing of dimensions, it matters whether a rectangle is increasing its height to the top or the bottom and its width to the left or the right for the stability score. Increasing the width to the left changes both the x-coordinate and the width while increasing the width to the right only change the width of the rectangle.

An example of why this is problematic is shown in Figure 2.3. In Figure 2.3a rectangle A is above rectangles B, C and D and the height of A is decreased which results in a stability score of $\sqrt{5^2} + 3\sqrt{5^2 + 5^2} \approx 26.2$. In Figure 2.3b rectangle A is below rectangles B, C and D and the height of A is decreased which results in a stability score of $3\sqrt{5^2} + \sqrt{5^2 + 5^2} \approx 22.1$. Figure 2.3a and Figure 2.3b are mirrored which intuitively means that the stability score should be equal in these two cases. Due to both the height and the y-coordinate being inside the root this is however not the case. The layout distance change function is thus not able to accurately reflect changes in position and shape in a conclusive way, as the score depends on the orientation of the rectangles.

The second major problem is that the groups of rectangles within the treemap are not considered in the layout distance change function. If a group of rectangles moves in its entirety, the layout distance score will penalize the distance moved for each rectangle separately while disregarding the group structure in the treemap. An example of this is given in Figure 2.4a.

In Figure 2.4a the sizes of rectangles A and B have been swapped, while the rectangles maintain the relative positions with regard to each other. As a result of this, all rectangles between A and B have a large change in their spatial position. The layout distance score penalizes every one of these position changes, which results in a relatively high stability score of $\approx 37.1$. The resulting layout is however in fact quite stable. The relative positions between the rectangles did not change by much and the group structures within the treemap stayed intact as well.

In the layout shown in Figure 2.4b the relative positions of the groups are no longer intact.
However, starting from the same layout as in Figure 2.4a, the resulting layout has a stability score of $\approx 34.4$ which is lower than the layout shown in Figure 2.4a. It should thus hold that the layout shown in Figure 2.4b is more stable but this does not appear to be true at all. This occurs due to the fact that the layout distance change function does not consider the adjacency or the more generalized relative positions. It thus has no possibility to consider group structures for the stability score.

Finally the layout distance change function has a minor problem, namely that the stability score is not normalized. A value of 90 can indicate both a high or a low stability depending on how large the initial rectangle is and how much rectangles there are in the treemap. Using the stability score given by the layout distance change function is therefore mostly useful to compare the stability score of two different possible layouts $L', L''$ from an original layout $L$. Using it to determine the stability score of a change from $L$ to $L'$ by itself becomes quite a lot harder. This problem can however easily be solved by normalizing the scores using the length of the diagonal of the input rectangle of the treemap.

### 2.2.2 Variants of the layout distance change function

Aside from the layout distance change function score there are three variants for the layout distance change function. The first variant is the variance of distance change function [20] as presented by Tak and Cockburn. The second variant is the centroid positioning measure [11] as presented by Hahn, et al. The third and final variant is the locational drift measure [20] as presented by Tak and Cockburn.
CHAPTER 2. STABILITY

Variance of distance change

The variance distance change function complements the layout distance change function by considering the variance of the layout distance change function.

\[
\text{VarianceOfDistanceChange}(L(t), L(t+1)) = \text{Var}(\sum_{r \in X} d(L(t, r), L(t+1, r)))
\]

Using the variance distance change function one can identify whether the score from the layout distance change function is the result of a large number of average movements or a small number of large movements.

While allowing more information to be conveyed through the stability score, the variance of distance change function however still does not address the problem that groups are not considered in the stability score. Both a large number of average movements and a small number of large movements can have a large impact on the group structures within the treemap. We thus still cannot identify whether the group structures remained intact or not.

The variance of distance change function moreover does not solve the problem of mixing of dimensions and the normalization problem as it still uses the layout distance change function as a basis.

Centroid positioning

The centroid positioning variant was presented by Hahn, et al. [11] and uses only the change in position of the centroid of a shape to determine the stability.

\[
\text{centroidStability}(L(t), L(t+1)) = \sum_{r \in X} d(c(L(t, r)), c(L(t+1, r)))
\]

While the centroid positioning score mixes the dimensions in a different way than the layout distance change function, it is still problematic. When the rectangle becomes equally smaller to the left and to the right the position of the centroid does not change while the rectangle did change. The rectangle would then be seen as completely stable while in practice it can change quite a lot. The change of the shape of the rectangle is thus not reflected in the stability score. An example of this is shown in Figure 2.5. In Figure 2.5a rectangle D and C are both stretched downwards which keeps the centroid of rectangle B unchanged. This results in a centroid positioning stability score of 4. In Figure 2.5b only rectangle D is stretched downwards. The centroid of rectangle B now does change. This results in a centroid positioning stability score of 4.5. Intuitively Figure 2.5b is more stable as Figure 2.5a as there are less changes to the treemap. However, the centroid positioning stability score of Figure 2.5a is lower than the centroid positioning score of Figure 2.5b which contradicts our intuition.
2.2. CURRENT DEFINITIONS

The centroid positioning score thus still suffers from the problem that is mixes the dimensions incorrectly. Moreover, it still does not encode the group structures in any way, and therefore it still suffers from the same problems as the layout distance change function.

(a) Rectangles D and C are both stretched downward. The centroid of rectangle B is unchanged.

(b) Rectangles D is stretched downward. The centroid of rectangle B has changed.

Figure 2.5: Figure 2.5a has a lower centroid positioning stability score than Figure 2.5b but is less stable.

Locational drift

The mean locational drift stability measure enhances the layout distance change function by no longer considering the stability within a single time iteration, but considering the stability over a larger period of time. It measures the average distance that each rectangle \( r \) is away from the center of gravity of rectangle \( r \) over the past \( y \) iterations. The center of gravity is defined as the average \( c(L(t-r), r) \) position over the past \( y \) iterations. The locational drift stability measure can then formally be defined as:

\[
locationalDrift(t) = \frac{1}{|R|} \sum_{r \in R} \frac{1}{y} \sum_{j=1}^{y} ||c(L(t-j, r), COG(L, r))||
\]

\[
COG = \frac{1}{y} \sum_{k=1}^{y} c(L(t-k, r))
\]

The locational drift measure thus additionally attempts to capture the stability over a larger period of time. However, as it uses the centroid of the rectangle to determine the distance, it suffers from the same problems as the centroid positioning stability measure. Moreover, it still does not encode the group structures in any way.

Another minor problem is that the locational drift measure as presented does not work when the period of time becomes extremely large. If the period of time becomes too large a rectangle might drift very slowly to another position. However, as the center of gravity is calculated over all iterations, it will on average have a large distance. This problem can
however be solved by calculating the center of gravity for a limited time period instead of over the entire time span.

For the locational drift measure a user study has been performed by Tak and Cockburn [20]. In this study, users which were highly familiar with treemaps were tasked to select a particular rectangle in the treemap. After each selection the layout was updated and the selection times were measured. The user study found a significant difference in selection times between a layout with a low locational drift and a random layout. This indicates that the drift over time of a rectangle should be taken into account for a complete definition of the perceived stability.

2.3 A new definition of stability

To better cover the notion of group structures in the layout with regard to the stability, we are going to develop a new stability measure. At the core of the measure we are going to use the relative positions of the rectangles with regard to the other rectangles in the layout. The more stable the relative positions of the rectangles are, the more stable the resulting stability score will be. By using the relative positions of rectangles with regard to one another, we are able to cover the notion of group structures. If group structures stay intact, then the relative positions of the rectangles within this group will stay roughly the same. The stability score will thus be higher than when the group structure would be destroyed as in this case the relative positions would be changed as well.

To determine the relative position of a rectangle B with regard to a rectangle A, we are going to consider in what general direction rectangle B is in with regard to rectangle A. To this end, we will divide the space around rectangle A in 8 sections $S = \{s_1, s_2, ..., s_8\}$. Section $s_1$ will represent the East, section $s_2$ the NorthEast, etc. as is shown in Figure 2.6.

![Figure 2.6](image)

Figure 2.6: The area around rectangle A is divided into 8 sections.

To determine the stability of the relative position between rectangles A and B in layouts $L(t)$ and $L(t + 1)$, we are going to calculate to what degree rectangle B stays in the same sections of rectangle A. We want to avoid large changes in the stability score when minor changes in the structure of the treemap occur. Therefore, we cannot simply use a binary score to determine the degree for whether it stays in the same section to determine the stability score. If we did use such a binary score we could for example have the follow-
2.3. A NEW DEFINITION OF STABILITY

ing problem. Consider that rectangle B is for 10% in section East and for 90% in section SouthEast in layout \( L(t) \) with regard to rectangle A. In layout \( L(t + 1) \) it has moved just far enough to be 100% in section SouthEast. Graphically this is shown in Figure 2.7. This would mean that when using a binary score, the resulting stability score would be high. However, the actual change in the structure is small and the stability score should thus be low.

![Figure 2.7: 10% of the area of rectangle B was in the NorthEast section of rectangle A. After a slight downwards movement 100% of the area of rectangle B is in the East section of rectangle A.](image)

To prevent this problem we are going to use the change in percentage that a rectangle is in each section. Let rectangle B be for 10% in the East section of rectangle A and for 90% in the Southeast section of rectangle A in layout \( L(t) \). Furthermore let rectangle B be for 100% in the SouthEast section of rectangle A in layout \( L(t + 1) \). The stability score will be equal to 0.1 as 10% of the area of the rectangle changed section.

We will denote the percentage that the rectangle \( L(t, B) \) is in section \( s_i \in S(r_i) \) of the rectangle \( L(t, A) \) by \( \text{percentage}(A, B, L, s_i) \) for all \( 1 \leq i \leq 8 \). We can then define the relative stability score between the two rectangles as follows:

\[
S_{\text{relative}}(A, B, L, L') = \frac{1}{2} \sum_{i=1}^{8} |\text{percentage}(A, B, L, s_i) - \text{percentage}(A, B, L', s_i)|
\]

It then remains to calculate the overall relative stability score. As rectangles that are not in both layout \( L(t) \) and \( L(t + 1) \) do not have a relative position with regard to each other, we will not consider these explicitly in the overall stability score. They are however considered implicitly, as the insertion and deletion of a rectangle will usually have an impact on the relative position of the rectangles that are present. To calculate the overall stability score we will use the average relative stability score for all combinations of rectangles A and B that are in both layouts:

\[
S_{\text{relative}}(L(t), L(t + 1)) = \frac{1}{|R|(|R| - 1)} \sum_{r, r' \in (R(t) \cap R(t+1)) \wedge r \neq r'} S_{\text{relative}}(r, r', L(t), L(t + 1))
\]

An example of the result of the application of this score is given in Figure 2.8. When calculating the stability score for Figure 2.8a using the relative stability score we obtain
a stability score of $\approx 0.136$. When calculating the stability score for Figure 2.8b using the relative stability score we obtain a stability score of $\approx 0.247$. The relative stability score of Figure 2.8b is thus significantly higher than the stability score of Figure 2.8a. This corresponds with the change in group structures in these two figures. The group structures in Figure 2.8a has changed significantly more than in Figure 2.8b.

Figure 2.8: Figure 2.8a is more stable than Figure 2.8b and has a lower relative stability score.

The new definition thus seems to achieve our goal of accounting for the group structures within the treemap. The new definition however does not take the form or the spatial position explicitly into account. To solve this one would require a mix of several stability measures that individually measure the change in form, the change in spatial position and the change in group structures. Each measure would then be assigned a weight factor to determine the influence of this factor. This weight factor should be determined through an user study. Determining these weight factors is however outside the scope of this thesis. For the remainder of this thesis we will therefore assume the relative stability score reflects the perceived stability adequately.
Chapter 3

Stable algorithm

To develop a treemap algorithm that can balance the stability score and the aspect ratios, we use the concept of local moves which will be presented in Section 3.1. We use these local moves in two new treemap algorithms. The first algorithm will be presented in Section 3.2 and is an incremental algorithm that works purely using local moves after the initial generation of the treemap. The second algorithm will be presented in Section 3.3 and is a hierarchical incremental algorithm that works using a combination of the approximation algorithm by Nagamochi and Abe [15] and local moves.

3.1 Local Moves

To develop our new algorithms we are going to use the concept of local moves. Local moves are manipulations of the layout that change the order equivalence graph of the layout only in a local section.

The idea to use local moves came from the twin binary sequence as presented by Young, et al. [24] They presented a method to traverse from any layout to any other layout given that the labeling of the rectangles did not matter. However, for our purposes the labeling of the rectangles does matter and we thus need to develop a new method.

The reason that we are going to manipulate the order equivalence graph using small changes in our algorithm is threefold. The first part of the reason is that it seems to hold that for small changes in the sizes of the rectangles, the relative position stability score is nearly equal to 0 when the order equivalence graph of the layout does not change. This assumption is validated in Section 4.1.1.

An example of this is shown in Figure 3.1. In Figure 3.1 a small change in the sizes of the rectangles occurs, while the order equivalence graph stays the same. The resulting structure of the treemap is almost identical to the original structure. More specifically, the
relative positions between the rectangles are almost identical which results in a low stability score. It seems that the order equivalence graph thus encodes the relative position up to a certain degree.

Figure 3.1: A number of small changes in the sizes occur, but the order equivalence graph does not change.

The second part of the reason is that we can manipulate the orderequivalence graph by a small amount at a time using local moves while being able to regenerate a valid layout after each step. These local moves furthermore do not have a large impact on the stability score. The validity of this statement is validated in Section 4.1.2. As the local moves have a limited impact on the stability, the local moves can be used as a tool to balance the stability with the aspect ratio. The more moves we are allowed to make, the more the aspect ratio will be able to approach the optimal at the cost of stability.

The third and final part of this reason is that we can reach all possible layouts using local moves as is shown in Section 3.1.2. We thus do not discard any possible layout. Specifically, it allows us to consider non-sliceable layouts in addition to sliceable layouts. Non-sliceable layouts have as the advantage that they have better lower bounds on the optimal maximum aspect ratio, as is shown in Appendix B.

3.1.1 Types of local moves

We will consider three local moves for our algorithm. The stretch move, the flip move and the edge flip move. The stretch move and the flip move are sufficient to reach all possible layouts as we prove in Section 3.1.2. The edge flip move is additionally considered, as it allows us to find non-sliceable layouts faster.

The stretch move

The stretch move stretches a rectangle A over a rectangle B. Let $ms$ be a maximal segment and let A and B be two rectangles adjacent to one of the endpoints of this segment. Without loss of generality we assume that $ms$ is a vertical maximal segment. If rectangles A and B do not have the same height we can apply a stretch move. Let rectangle A denote the rectangle with the smallest height and without loss of generality assume that rectangle A is to the left of $ms$. 

26 Stability of treemap algorithms
3.1. LOCAL MOVES

To apply the stretch move we then stretch rectangle A over rectangle B as is graphically displayed in Figure 3.2. After applying the stretch move we update the layout using the algorithm presented in Section 1.2.4 to make sure the areas of all the rectangles have the correct size again.

![Figure 3.2: A stretch move is applied on the maximal segment ms to stretch rectangle A over rectangle B.](image)

**Flip move**

The flip move flips rectangle A and rectangle B from being horizontally adjacent to vertically adjacent and vice verse. Let *ms* be a one-sided maximal segments with exactly two rectangles adjacent to it. Let A and B be the two rectangles adjacent to the maximal segment *ms*. Note that in this case, rectangles A and B together form a rectangle as well. Without loss of generality we assume that *ms* is a vertical maximal segment and that rectangle A is to the left of *ms*.

To apply the flip move we then place rectangle A below rectangle B in the rectangle formed by the rectangles A and B which is graphically displayed in Figure 3.3. In contrast to the flip move it is not required to recalculate the layout, as the flip move does not change the areas of any rectangles in the layout.

![Figure 3.3: A flip move is applied on the maximal segment ms to flip rectangles A and B.](image)

**Edge flip move**

The edge flip move flips the direction of an edge split. An edge flip can be performed when two maximal segments *ms*₂, *ms*₃ have an endpoint on both sides of *ms*₁. Moreover these endpoints should not be at the endpoints of *ms*₁. The edge flip move then merges the maximal segments *ms*₂ and *ms*₃ into one maximal segment, and breaks *ms*₁ into two maximal segments. This is graphically shown in Figure 3.4.
CHAPTER 3. STABLE ALGORITHM

Figure 3.4: An edge flip move is applied on the maximal segment $ms$ to flip rectangles $A$ and $B$.

After the merging and the splitting of the maximal segments, the adjacencies of the maximal segments to the rectangles can be uniquely determined. This allows us to update the order equivalence graph, which in turn can be used to regenerate the layout such that each rectangle has the correct area using the algorithm presented in Section 1.2.4.

As was mentioned before this move is used to be able to find non-sliceable layouts faster. In Figure 3.5 an example of the edge flip move being faster than the stretch and flip moves is shown. Using only the stretch and flip moves means that we require at least 4 moves to be able to go from the original layout to the non-sliceable layout as is shown in Figure 3.5a. However, using the edge flip move this is possible using a single move as is shown in Figure 3.5b.

Figure 3.5: A comparison between the number of moves required to go from a specific sliceable layout to a windmill layout with and without edge flips.
3.1. LOCAL MOVES

3.1.2 All rectangular layouts are reachable using local moves

We will now prove that we can reach all possible layouts using a series of flip and stretch moves.

**Theorem 1.** Using only stretch and flip moves it holds that for any two layouts \( L, L' \) consisting of the same set of rectangles \( R \) there exists a sequence \( S \) of stretch and flip moves such that \( G \) is transformed into \( G' \).

We will prove Theorem 1 using three lemmas.

The first lemma used states that we can reach a stack layout from any layout \( L \).

**Lemma 2.** From any layout \( L \) we can reach a vertical stack layout \( L \) using only Flip and Stretch moves.

The second lemma used states that we can turn a stack layout into a sorted stack layout.

**Lemma 3.** From a vertical stack layout \( L \) we can reach a sorted vertical stack layout \( L' \) using only flip moves.

The third lemma used states that it is possible to invert all the moves. That is, for each move \( x \) there exists a series of moves \( Y \) such that performing the moves in \( Y \) after performing move \( x \) on any layout \( L \) results in the same layout \( L \). We will denote the inverse of a move \( x \) as \( x^{-1} \).

**Lemma 4.** The Flip move and the Stretch move are invertible.

If these three lemmas are all valid, then we can proof Theorem 1 as follows:

Let \( S_1 \) denote the sequence of local moves to go from the layout \( L \) to the stack layout. Let \( S_2 \) denote the sequence of local moves to go from the stack layout to the sorted stack layout. Let \( S_3 \) denote the sequence of local moves to go from the layout \( L' \) to the sorted stack layout.

The sequences \( S_1, S_2, S_3 \) must all exist due to the Lemma 2 and Lemma 3. As it is possible to invert a move due to Lemma 4, it is also possible to invert a sequence. Let \( S_3 = (move_1, ..., move_i, ..., move_n) \) be a sequence that must be inverted. Let \( S_3^{-1} \) denote the inverted sequence which is constructed by inverting the order and inverting all the moves of the sequence \( S_3 \). For \( S_3^{-1} \) to be the inverse of \( S_3 \) it must hold that \( S_3 \) followed by \( S_4 \) must be equivalent to not performing a move at all:

\[
S_3; S_3^{-1} = (move_1, ..., move_i, ..., move_n); (move_n^{-1}, ..., move_i^{-1}, ..., move_1^{-1}) = \emptyset
\]

Thus \( S_3^{-1} \) is indeed the inverted sequence of \( S_3 \). \( S_3^{-1} \) thus contains the sequence to go from the sorted stack layout to the layout \( L' \).
CHAPTER 3. STABLE ALGORITHM

To transform \( L \) into \( L' \) we perform the moves in the sequence \( S = S_1; S_2; S_3^{-1} \) which transform the layout \( L \) in succession to a stack layout, the sorted stack layout, and the layout \( L' \).

We will now proof that the three involved lemmas are indeed valid.

Reaching the vertical stack layout

To prove Lemma 2 we need to show that is it possible to reach the stack layout from any layout \( L \). We define a vertical stack layout as a layout \( L \) that has does not contain any inner vertical maximal segment. An example of such a layout is shown in Figure 3.6. Thus to transform layout \( L \) into a stack layout, we only need to make sure that we have removed all inner vertical maximal segments.

![Figure 3.6: An example of an unsorted vertical stack layout.](image)

We will pick an inner vertical maximal segment \( ms \) and we will iteratively remove adjacent rectangles from \( ms \), until \( ms \) has exactly two rectangles adjacent to it. Let A denote the rectangle adjacent to the left top of \( ms \) and let B denote the rectangle adjacent to the right top of \( ms \). To remove a rectangle from \( ms \) we have two cases:

Case 1: The height of rectangle B is unequal to the height of rectangle A. Without loss of generality we will assume that rectangle A has a greater height then rectangle B. In this case we use a stretch move to stretch rectangle B over rectangle A as is shown in Figure 3.7a. Rectangle B is then no longer adjacent to \( ms \) and we have not introduced a new maximal vertical segment.

Case 2: Rectangle A is equal in height to rectangle B. In this case we use a flip move to flip rectangles A and B as is shown in Figure 3.7b. Rectangles A and B are then no longer adjacent to \( ms \) and we have not introduced a new maximal vertical segment.

As an inner maximal segment must have a rectangle on both sides of the segment at the endpoint of the segment, one of these two cases must always hold. Moreover, as we remove at least one adjacent rectangle from \( ms \) in each case, it holds that eventually \( ms \) will have exactly 2 rectangles adjacent to it.

When \( ms \) only has 2 adjacent rectangles, we will remove the maximal segment \( ms \) completely by performing a flip move on the two remaining rectangles as is shown in Figure 3.8. We have thus removed a vertical maximal segment without introducing any new vertical segments. The total number of vertical maximal segments is thus reduced by 1.
3.1. LOCAL MOVES

(a) Rectangle B is stretched over rectangle A.

(b) Rectangle A and B are flipped using a flip move.

Figure 3.7: The two cases for removing a rectangle from $ms$. The dotted lines represent the other rectangles that might be adjacent to $ms$.  

Figure 3.8: Rectangles A and B are flipped using a flip move. $ms$ is now removed.

By repeatedly using removing a vertical maximal segment using this procedure, we can remove all inner vertical maximal segments from $L$ to reach a vertical stack layout. Therefore Lemma 2 holds.

Reaching the sorted vertical stack layout

To prove Lemma 3 we need to show that is it possible to reach the sorted vertical stack layout from any vertical stack layout $L$.

To transform the vertical stack layout $L$ into the sorted vertical stack layout, we will use BubbleSort on the vertical stack layout. To be able to use BubbleSort on the vertical stack layout, we have to be able to swap two adjacent elements. Two adjacent rectangles A and B in a stack layout can be swapped by applying two flip moves on rectangles A and B as is shown in Figure 3.9. We can thus use BubbleSort to reach the vertical stack layout and Lemma 2 holds.

Figure 3.9: A and B are swapped using flip moves.
CHAPTER 3. STABLE ALGORITHM

Inverting the moves

To prove Lemma 4 we need to show that it is possible to invert the Flip move and the Stretch move.

We can invert a flip move on rectangles A and B by performing three flips moves on rectangles A and B as is shown in Figure 3.10a.

We can invert a stretch move from rectangle A over rectangle B by performing a stretch move from rectangle B over rectangle A as is shown in Figure 3.10b.

![Inversion Diagram](image)

(a) Inverting the flip move. (b) Inverting the stretch move.

Figure 3.10: Inversions of the flip move and the stretch move.

As we can invert both the flip move and the stretch move Lemma 4 thus holds.

3.2 Incremental moves algorithm

The incremental moves algorithm starts by generating an initial treemap using the approximation algorithm by Nagamochi and Abe [15]. After the generation of the initial treemap, we will only use local moves to change the treemap in all following iterations.

For every iteration we are going to use a bounded breadth first search approach to find the layout that has the best maximal aspect ratio within \( k \) local moves. To find all possible layouts \( L' \) reachable from a layout \( L \) using local moves we are going to go over each inner maximal segment \( ms \in MS(L) \). For each of these maximal segments we can either perform a flip move, or we can perform two stretch moves. If \( ms \) is horizontal we can perform a stretch move on the left endpoint of \( ms \) and a stretch move on the right endpoint of \( ms \). If \( ms \) is vertical we can perform a stretch move on the top endpoint of \( ms \) and a stretch move on the bottom endpoint of \( ms \). If edge flips are additionally considered then for each maximal segment, we additionally have the possibility to perform two edge flip moves on the endpoints of the maximal segments.

After the first iteration we will no longer consider all inner maximal segments \( ms \in MS(L) \). Instead we will restrict ourselves to the maximal segments \( ms \) that are in the neighborhood \( N \) of the local move that was performed. iteration. The neighborhood \( N \) is
defined as the maximal segments $ms \in MS(L)$ for which the adjacency to the rectangles in $\mathcal{R}(t)$ has changed. By restricting ourselves to this neighborhood we drastically reduces the total search space which allows us to handle larger amount of rectangles. Moreover, we still consider all maximal segments that are close to the rectangle that we are optimizing, as these are the most promising candidates to improve the aspect ratio.

Restricting ourselves to considering only the neighborhood has the negative consequence that for larger layouts, we can only optimize the aspect ratios of a few rectangles. To make sure that we can optimize multiple rectangles with a large aspect ratio in a single timestep, we repeat the above procedure $x$ times.

The depth of the breadth first search in the algorithm is bounded for two reasons. The first reason is that by bounding the depth, we restrict the amount that we will change the structure of the treemap. As there are few changes the resulting stability score will also be low. The second reason it that going through the entire search space is practically impossible. If there are 10 rectangles in the layout, the complete search space already equals $\approx 1.09 \times 10^{20}$ nodes. Therefore, we need to keep the search space limited out of necessity.

### 3.2.1 Approximation algorithm

The approximation algorithm that we will be using to generate the initial treemap is the approximation algorithm as presented by Nagamochi and Abe [15]. It works by sorting the input from low to high and recursively dividing the sorted list into two groups of roughly the same size.

The algorithm is guaranteed to give a treemap which has a maximal aspect ratio of at most $a(R_0), 3, 1 + \max_{i=1,\ldots,|\mathcal{R}(t)|} \frac{s(r_{i+1})}{s(r_i)}$ where $r_{i+1} \geq r_i$. It thus bounds the maximal aspect ratio by the aspect ratio of the input rectangle, a constant value of 3 and the maximum ratio between the sizes of two consequential rectangles in the size-sorted list of rectangles.

The approximation algorithm on itself is very unstable as the input needs to be sorted, and furthermore it determines the orientation of the split based on the orientation of the previous recursion.

Requiring the input to be sorted has the effect that a minor change in size can drastically influence the layout. After sorting a rectangle might now be in the leftmost part of the treemap instead of the rightmost part. Deciding the orientation of a slice based on the best aspect ratio has the effect that a minor change in the size can flip the orientation at every level. This can result in a very different treemap. These two problems together make sure that we can have a completely different treemap, even if there are only small changes in the sizes.

However, it is quite suitable for use as an initial algorithm as it is the only algorithm which guarantees an upper bound on the maximal aspect ratio. The aspect ratio of the
CHAPTER 3. STABLE ALGORITHM

initial treemap is thus always bounded. For most practical applications the aspect ratio of
the input rectangle is low. This leaves only the distribution of the size of the rectangles as a
potential problem. We have assumed that this will not be a problem, but if the distribution
has a very high variance, a different algorithm might be better suited to generate the initial
treemap.

3.2.2 Handling deletions and additions

In order for our algorithm to be usable in practice, it should also be able to handle inser-
tions and deletions in the data.

Handling deletions

Let \( L \) be the current layout, and let \( L' \) be the layout that we need to generate. To handle
the deletions of rectangles that are present in layout \( L \) but not in \( L' \), we will delete them
one by one from the layout \( L \) before generating the layout for \( L' \).

Let rectangle \( A \) denote a rectangle we are going to delete from the layout \( L \). If rectangle
\( A \) is a grounded rectangle in \( L \), then by Lemma 5 we know that there exists a one-sided
maximal segment \( ms \) in layout \( L(t) \) such that \( A \) is the only rectangle adjacent to one side
of \( ms \). To remove rectangle \( A \) from \( L \) we are going to stretch all rectangles on the other
side of \( ms \) over rectangle \( A \). This removes rectangle \( A \) from the layout. This is graphically
shown in Figure 3.11.

![Figure 3.11: Deleting a grounded rectangle A from a layout.](image)

If rectangle \( A \) is not a grounded rectangle in \( L \), then we cannot simply delete it using
these stretch moves. Removing rectangle \( A \) directly would leave a hole in the layout as is
shown in Figure 3.12 which we are not able to fill directly using stretch moves.

![Figure 3.12: Deleting a rectangle A that is not grounded leaves a hole which cannot be
directly filled with stretch moves.](image)
3.2. **INCREMENTAL MOVES ALGORITHM**

To be able to remove rectangle $A$ we are going to transform the layout $L$ such that rectangle $A$ becomes a grounded rectangle. To do this we are going to repeatedly apply stretch moves to rectangle $A$ before deleting it. There are two cases to consider for this:

The first case is that at least one side $s$ of rectangle $A$ is adjacent to exactly one rectangle which we will denote as rectangle $B$. As rectangle $A$ is not a grounded rectangle, it holds that the side $s$ is at the endpoint of a maximal segment $ms$ by Lemma 7. Moreover, it must hold the rectangle $A$ is longer alongside $ms$ than rectangle $B$. If this was not the case then rectangle $A$ would either be a grounded rectangle or there would be more than one rectangle adjacent to $A$ on this side. It must thus be possible to stretch rectangle $A$ over rectangle $B$. The result of this is shown in Figure 3.13.

![Figure 3.13: Rectangle A is stretched over rectangle B in an non-sliceable layout.](image)

The second case is that no side $s$ of rectangle $A$ is adjacent to exactly one rectangle. We then pick a maximal segment $ms$ for which it holds that rectangle $A$ is adjacent to it. It must then hold that there exists a rectangle $B$ adjacent to $ms$ which we can stretch over rectangle $A$. We repeatedly find such a rectangle $B$ and stretch it over rectangle $A$ until there is only one rectangle adjacent to rectangle $A$ over this maximal segment. We will then follow the procedure for the first case. This result of this process is shown in Figure 3.14.

![Figure 3.14: Rectangles B and F are stretched over rectangle A. Afterwards rectangle A is stretch over rectangle G.](image)

We can thus always stretch rectangle $A$ of the maximal segment $ms$. By repeatedly performing this in one direction, rectangle $A$ will eventually become a grounded rectangle. Either a side of $s$ will become one-sided with $A$ on the one-sided side, or it will become adjacent to the boundary rectangle. If rectangle $A$ is adjacent to the boundary, rectangle $A$ must be a grounded rectangle as well by Lemma 6. After transforming rectangle $A$ to a grounded rectangle we delete rectangle $A$ using the process for grounded rectangle which is explained above.

Alternatively, the second case can also be handled using a single edge flip move. We pick any maximal segment $ms_1$ that $A$ is adjacent to. Without loss of generality we assume...
that $ms_1$ is to the bottom of rectangle A. As rectangle A is not a grounded rectangle and rectangle A is not adjacent to exactly one rectangle on any side, it holds that $ms_1$ must have at least two rectangles adjacent to either side.

Let C denote the rectangle adjacent to rectangle A and adjacent to $ms_1$ on the same side as A. Without loss of generality we assume that C is to the right of A. Let $ms_2$ denote the maximal segment between rectangles A and C.

As rectangles A and C are both adjacent to $ms_1$, $ms_2$ has an endpoint on $ms_1$ and this endpoint is not on one of the endpoints of $ms_1$. As $ms_1$ has at least two rectangles on the other side which we will denote as F and G, there must exist a maximal segment $ms_3$ such that an edge flip can be performed using the maximal segments $ms_1$, $ms_2$ and $ms_3$. Without loss of generality we assume that rectangle F lies to the left of $ms_2$ and rectangle G lies to the right of $ms_2$. By performing this edge flip, rectangle A will become a grounded rectangle as is shown in Figure 3.15.

Figure 3.15: An edge flip is performed with maximal segments $ms_1$, $ms_2$ and $ms_3$. Rectangle A then becomes a grounded rectangle.

Handling additions

Let $L$ be current layout and let $L'$ be the layout that we need to generate. To handle the additions of the rectangles that are present in layout $L$ but not in $L'$ we will add them after the generation of the layout $L'$ without the additional rectangles.

To add rectangle A to the layout $L'$, we are going to split an existing rectangle B into rectangle A and rectangle B as is shown in Figure 3.16. We will then recalculate the positions of the rectangles in the layout using the algorithm as explained in Section 1.2.4.

To decide which rectangle we are going to use as rectangle B, we are going to consider all rectangles present in the layout. We will pick the rectangle which results in the best possible maximal aspect ratio after inserting rectangle A in the layout.

Figure 3.16: Rectangle B is split into rectangles B and A to insert rectangle A in the layout.
3.3 Hierarchical incremental moves algorithm

The main downside of the incremental moves algorithm is that the depth of the search is limited by the size of the search space. The hierarchical incremental moves algorithm aims to remedy this by transforming the single-level treemap into a hierarchical structure. The amount of children for each node in this structure will be limited. Because the number of children will be limited, it is possible to search through a larger number of moves for every node in the structure which increases the speed drastically. However, it loses the power of allowing all possible layouts as the rectangles are now put into a hierarchy.

The algorithm works as follows:

We will start the algorithm by generating a hierarchical structure from the rectangles in the treemap. The rectangles are presented by leafs and groups of rectangle are represented by internal nodes. Each internal node will have a size equal to the sum of the sizes of its children.

Using this structure we will generate the initial treemap. Starting from the root and recursing downwards, we will generate a local layout for each internal node using the approximation algorithm as explained in Section 3.2.1. We will use the size of the children as the size of the rectangles, and the rectangle associated to this internal node as the input rectangle for the layout. As we are recursing downwards each internal node will have a rectangle associated to it from the approximation algorithm in the parent node. For the root node the input rectangle $R_0$ will be used.

After the initial generation, we will attempt to maintain the maximal aspect ratios in the treemap at each time step. If the aspect ratio of a rectangle $A$ crosses a threshold $c_1$ we will attempt to fix the aspect ratio. We will find the first group ancestor $g$ of rectangle $A$, for which it holds that we can improve the aspect ratio of its children to less than $c_2$ using a combination of local moves and the approximation algorithm. We will regenerate the layout for $g$ and recurse downwards in its children. For each child $c$ where at least one of the rectangles has an aspect ratio larger than $c_2$, we will try to improve the layout using a combination of local moves and the approximation algorithm.

We will now explain each step of the algorithm in detail.

3.3.1 Generating the initial treemap

To generate the initial treemap, we start by generating the hierarchical group structure.

Generating the hierarchical group structure

We will start by putting all the rectangles $r_i \in \mathcal{R}(t)$ into a leaf group and collect all these leaf groups into a set of groups $\mathcal{X}$. We let the group that only contains the rectangle $r_i$ be
CHAPTER 3. STABLE ALGORITHM

denoted by \( g(r_i) \). \( c(g) \) will denote the direct children of a group \( g \). \( C(g) \) will denote all
the descendants of a group \( g \) and \( R(g) \) will denote the input rectangle of group \( g \).

We are now iteratively going to join together the groups in \( X \) to create groups containing
at most \( k \) items. We will do this as follows:

We will start by determining how many groups we require such that no group has more
than \( k \) elements:

\[
z = \left\lceil \frac{|X|}{k} \right\rceil
\]

Let \( \mathcal{Y}(j) \) denote the new group that will be create where \( 1 \leq j \leq z \). We are then going to
distribute each group in \( X \) to one on the group \( \mathcal{Y}(j) \) using a greedy algorithm to balance
the sizes of the groups.

To do this we are going to sort \( X \) based on the size of the groups. Let \( X(i) \) denote the \( i \)’th
largest group in \( X \). We will then iterate over \( i \) from 1 to \( |X| \), and add \( X(i) \) to the parent
group \( \mathcal{Y}(j) \) where \( j \) follows the sequence \( \{1, 2, \ldots, z-1, z, z-1, \ldots, 2, 1, 1, 2, \ldots\} \).

After we have added all the groups \( g \in X \) to parent groups \( p \in Y \), we will let \( X = Y \) and
divide \( X \) into groups again. We will repeat doing this until there is only one group left in
\( X \) which will be the root group.

Generating the initial treemap from the group structure

After the group structure is generated we will generate the initial treemap recursively
using the approximation algorithm. We do this as follows:

As an input we have a set of rectangles \( R \), the input rectangle \( R_0 \) and the maximal amount
of children per group \( k \). Let \( g \) denote the root of the structure, and set the input rectangle
of this group \( R(g) \) to \( R_0 \). We will then calculate the layout \( L' \) for the children of this graph
using the approximation algorithm with input rectangle \( R(g) \) and using \( c(g) \) as the list of
rectangles. For each of the children \( c \in c(g) \) we will then set \( R(c) \) equal to the rectangle
\( L'(c) \). We will then recurse into the children.

When the recursion is finished all the groups will have a rectangle associated to them. To
generate the initial treemap we will finally go through all the rectangles \( r_i \in R \) and set
\( L(0, r_i) = R(g(r_i)) \).

3.3.2 Maintaining the treemap

For the consecutive iterations we will not regenerate the treemap from scratch as all cur-
rent treemap algorithms do, but we will instead use the existing treemap as a basis.

We will first update all the sizes of the rectangle \( r \in R \) in the treemap and update the
layout using the algorithm presented in Section 1.2.4. We will then update the associated
rectangles \( R(g) \) for each group \( g \) by letting \( R(g) \) be the enclosing rectangle of all leaf descendants of \( g \).

After the group structure is updated we will attempt to fix the aspect ratios in the treemap. Let \( c_1 \) be a constant indicating the maximum aspect ratio the children of a group are allowed to have before we are going to attempt to fix it. Let \( c_2 \leq c_1 \) denote the maximum aspect ratio the children of a group are allowed to have after improving it. \( c_2 \) is lower than \( c_1 \) to make sure that we only perform changes that improve the aspect ratio enough in order to keep the stability score as low as possible.

As we are working with a hierarchical structure it is not always possible to improve the aspect ratios directly at a group. Moreover, changing the layout at a parent group changes the aspect ratios of all descendant groups. We will therefore identify the highest level of groups which we need to change to improve the aspect ratio.

We start by identifying which leaf groups have a bad aspect ratio. Let \( X \) denote the set of leaf groups \( g \) for which the aspect ratio \( a(g) \) is larger than \( c_1 \).

We then check how far upwards we must go into the group structure before we can improve the aspect ratio of the groups \( g \in X \).

For each parent group \( p = p(g) \) of the groups \( g \in X \) we will check if the aspect ratios of its children are above \( c_1 \) and whether it is possible to improve the aspect ratios to below \( c_2 \). To determine whether it can be improved enough we will first check if we can improve the aspect ratios enough using only local moves as this is the most stable option. If this is not possible we will first calculate a new layout using the approximation algorithm and afterwards use local moves to improve the layout.

The improvement using only local moves will be done in a similar way as was done for the incremental algorithm presented in Section 3.2. The only change will be that we will stop the search as soon as we found a layout where the aspect ratio of all rectangles is below \( c_2 \). If the aspect ratios were bad enough and we can improve them enough we will add \( p \) to the set \( Y \). If the aspect ratio were bad but we could not improve them enough we will recurse in the parent of \( p \) to find a group where the aspect ratios are bad and we can improve it enough.

We have now identified the highest level nodes that we need to change to improve the aspect ratios. However, as the group structure is a tree structure and we want to change the layout of each group at most once, a number of groups should be removed as these will be changed higher in the tree. We therefore remove all groups \( g \) from \( Y \) for which it holds that an ancestor of \( g \) is in \( Y \).

Finally we can update the layout of the groups. Starting from the groups in \( Y \) and recursing downwards we will check if the aspect ratios of its children are below \( c_1 \) and whether we can improve it to below \( c_2 \). If this is the case then we will perform the improvement. The layout \( L(t+1) \) can now be generating by using the positions of the groups \( g(r) \) associated to the rectangles \( r \in \mathcal{R}(t+1) \).
CHAPTER 3. STABLE ALGORITHM

3.3.3 Handling deletions and additions

For this algorithm we have to be able to handle deletions and additions in the data as well. Additions and deletions will be handled in a similar way as they are handled in the incremental moves algorithm presented in Section 3.2. There are however three differences.

The first difference is that when removing a rectangle \( r \) we only need to consider the layout of the first ancestor \( p \) of \( g(m) \) that has more than 1 child instead of the entire layout. Considering the entire layout is not needed as we know that the children of \( p \) are enclosed by a rectangle. We thus only need to consider the rectangles in \( p \) for the series of stretch moves that might need to be performed to make \( r \) an grounded rectangle.

The second difference is that when removing and adding a rectangle we need to update the group structure as well. How we are updating the group structure will be explained below.

Finally, when adding a rectangle \( r \) we can no longer choose from all possible rectangles where to insert it. Instead we are restricted to the rectangles inside the parent \( p \) of the group \( g(r) \) that \( r \) is added to.

Handling deletions

When a rectangle \( r \) gets deleted from the input data it changes the size of every group it is contained in. We will go through all ancestors \( p \) of \( g(r) \) and update the sizes of the groups. If \( p \) contains \( g(r) \) as a direct child, we will remove \( g(r) \) from the children. If \( g(r) \) was the only child of \( p \), we will delete \( p \) in its entirety.

Handling additions

To handle the additions we are going to use the concept of the first level groups. The first level groups are defined as the parent of the leaf groups.

When a rectangle \( r \) get added to the treemap we have to determine in which first level group we will add the rectangle. We will not insert rectangles at a ancestor \( p \) of a first level group. If we would do this, then \( p \) would have one child consisting of a single rectangle, and at least one child that consists of multiple rectangles. If we assume that the sizes of the rectangles fluctuate around a common mean, then this would mean that group consisting of a single rectangle is almost always significantly smaller than the rest of the groups. This discrepancy in sizes makes it hard to obtain low aspect ratios.

Moreover, the most volatile changes in the group structure will happen at the lower levels of the group structure. In the higher levels a group \( g \) consists of a larger amount of rectangles. A low size in a child of \( g \) can be balanced out by a high value in a child \( g \). This makes sure that in the higher levels, the value will be closer to the mean. In the bottom
levels there is less chance for this averaging to occur and these levels will thus be more volatile.

To make sure that this averaging occurs as much as possible, we attempt to keep the number of rectangles in the bottom levels balanced. When adding a rectangle $r$ we will thus try to add it in such a way that the groups become balanced.

We first find the smallest first level group $g$ which has less than $k$ items in it. If such a group $g$ exists then we will add the group $g(r)$ as a child to the group $g$. If no such group $g$ exists, then all first level groups have exactly $g$ children in them. For each of the first level groups $g$ we will then add another level as follows:

Given is a first level group $g$. For each of the children $c \in c(g)$ of $g$ we will replace the child $c$ with a new group $g'$ which has exactly one child which equals $c$. After adding a level to each of the first level groups we will then again find the smallest first level group $g$ which has less than $k$ items in it. This group must now exist and we will add the group $g(r)$ as a child to the group $g$. 

Stability of treemap algorithms
Chapter 4

Experimental evaluation

In this section we are going to investigate two problems through experimental results. The first problem is how the local moves correspond to the relative stability score. In Section 3.1 we had assumed that the order equivalence graph encode the relative position of the rectangles up to a certain degree. In Section 4.1.1 we will determine if this is indeed the case. Moreover, we had additionally assumed that performing a local moves has a limited impact on the relative stability score. In Section 4.1.2 we will determine if this is indeed the case.

The second problem is to determine how the newly developed algorithms perform in comparison to the existing algorithms. In Section 4.2 we will determine this through experiments on both artificially generated and real data.

4.1 Correspondence of local moves with stability

To determine how the local moves correspond with the stability we are going to perform two experiments. The first experiment will determine the relation between order equivalence of two layouts and the relative stability score. In particular we will test whether it is true that when two layouts are order equivalent and the change in sizes in the rectangles is small, the relative stability score is low. The second experiment will determine the average impact of each local move on the relative stability score. It will test the assumption that since the local moves only change the order equivalence graph of a layout slightly, the relative stability score will be low as well.
4.1.1 Order equivalence and stability

We will now determine the relation between order equivalence of two layouts and the relative stability score.

To perform the experiment we will first generate the initial entry of the dataset we will generate between 5 and 25 rectangles. The size of these rectangles will be distributed according to an uniform distribution in the interval $[1, 100]$. After the initial entry, we will incrementally change the sizes of the rectangles in the treemap. We will change the size of a rectangle $r$ randomly by a real number between $-x$ and $x$ for each timestep. The values of $x$ that will be considered are 5, 10, 25, 50 and 100. The size of the rectangle $r$ will remain restricted between 1 and 100.

We will then generate the initial treemap using the Approximation algorithm [15] as presented by Nagamochi and Abe which is explained in detail in Section 3.2.1. We will change the sizes of the rectangles a 100 times. After changing the sizes we will update the treemap with the new sizes using the algorithm presented in Section 1.2.4. We will then calculate the relative stability score. Finally, after calculating the relative stability score we will perform the local move that results in the treemap with the minimal maximum aspect ratio to generate a new treemap. This makes sure that we encounter a larger variety of different treemap. We will repeat this experiment a 100 times to make sure that the results are not influenced significantly be randomness.

The results of this experiment are graphically displayed in Figure 4.1. As can be seen the relative stability score are quite low. Even when the values can change by 100 which means that the values are completely randomized in each step the score is still below 0.2 as long as the order equivalence graph stays the same. It thus seems to hold that when two layouts are order equivalent, the relative stability score is low. The smaller the changes in the sizes of the rectangles are, the lower the resulting relative stability scores will be.

![Figure 4.1: The average relative stability scores when the sizes of the rectangles change while the layouts remain order equivalent.](image)

Stability of treemap algorithms
4.1. CORRESPONDENCE OF LOCAL MOVES WITH STABILITY

4.1.2 Local moves and stability

To determine the average impact of each local move on the relative stability score we are going to use a similar setup as the previous experiment. However, before performing the local move to update the treemap, we will calculate the stability score for each possible move that can be performed on the layout. We will aggregate the stability scores by move types such that the average impact of each type of local move becomes clear.

The results of this experiment are graphically displayed in Figure 4.2. The change in the relative stability score seems to be quite low if only a single local move is performed. The average impact of each type of local move is thus very low and the local moves indeed change the relative stability score by only a very limited amount.

![Figure 4.2: The average relative stability scores when a local move is performed for each type of local move.](image)

Stability of treemap algorithms 45
CHAPTER 4. EXPERIMENTAL EVALUATION

4.2 Performance of treemap algorithms

To evaluate the new treemap algorithms we are going to benchmark it on a number of datasets against the existing treemap algorithms. We are going to benchmark them in terms of the maximum aspect ratio, the average aspect ratio, the relative stability score and the layout distance change function score. The data used to benchmark the algorithms will be both artificially generated datasets and real datasets. Using artificial datasets we can generate a large amount of data to compare the different algorithms, and we can determine advantages and disadvantages of the different algorithms. The real datasets give additional insight on practical problems that may occur with algorithms and give examples of how the algorithms perform in practice.

4.2.1 Artificial datasets

We will generate the artificial datasets using a basic set of rules. For each experiment we will change a single variable to see what the influence on the resulting treemaps are. The basic set of rules is as follows:

To generate the initial entry of the dataset we will generate between 5 and 25 rectangles. The size of these rectangles will be distributed according to an uniform distribution in the interval $[1, 100]$. After the initial entry, we will incrementally change the sizes of the rectangles in the treemap. We will change the size of a rectangle $r$ randomly by a real number between $-5$ and $5$ for each timestep. The size of the rectangle $r$ will remain restricted between 1 and 100.

The experiments that we will be running modify the basic set of rules as follows:

1. No modification, we will use the basic set of rules as a baseline experiment. We will use the baseline experiment to compare with the rest of the experiments.
2. To test the influence of the number of rectangles on the algorithms, we will change the maximum number of rectangles that can be in the treemap from 25 to 50.
3. To test the influence of the size of the rectangles on the algorithms, we will change the maximum sizes of the rectangles from 100 to 1000.
4. To test the influence of the stability of the sizes of the rectangles, we will change the amount the sizes of the rectangles can decrease and increase from -5 and 5 to -25 and 25.
5. To test the influence of additions and removals of rectangles in the treemap, we will randomly add and remove rectangles from the treemap. For each rectangle $r$ in the layout $L(t)$, there is a 10% chance that it will be removed from the layout in $L(t + 1)$. We will furthermore add up to $25 - |R(t)|$ new rectangles $r'$ to the layout $L(t + 1)$. 

Stability of treemap algorithms
Each of these rectangles $r'$ has a chance of 10% that it will be added to the treemap. The initial size of these rectangles will be according to a uniform distribution in the interval $[1, 100]$.

6. To test the influence of the distribution of the sizes of the rectangles on the algorithms, we will change the distribution used. Instead of generating the initial entry using a uniform distribution we will generate the initial entry to a log-normal distribution. Furthermore the size of each rectangles $r$ in $L(t)$ will be changed at each step by multiplying the current size $s(r)$ by a factor $e^x$. $x$ is randomly drawn from a normal distribution with mean 0 and standard deviation 0.05. This method has been used before to evaluate the performance of treemap algorithms over time by Bederson et al. [2] and simulates a log-normal random walk. Moreover, the log-normal distribution is a heavy-tailed distribution that is common in naturally occurring positive-valued data [16]. It should thus be a good indicator of the overall quality of each treemap algorithms.

We will run each experiment 100 times for 100 iterations per algorithm to minimize the effect of randomness.

4.2.2 Real datasets

In addition to the generated datasets we will use two real world datasets. For both datasets we will use the complete dataset available which means that insertions and deletions will occur in both datasets. We will collect data on the average aspect ratio, the maximal aspect ratio, the layout distance change function score and the newly developed relative stability score.

The first real world dataset shows the popularity of first names for newborns in the Netherlands per year from 1993 until 2014. The data is trimmed to show only the 100 most popular male and female names per year and is obtained from the Meertens Instituut KNAW [14]. The data consists of a total of 451 unique names which appear in the data between 1993 and 2014.

The second real world dataset shows the audience rating per channel on the dutch television per year from 2004 until 2015. The data is obtained from the annual reports of Stichting Kijkonderzoek [19]. The data consists of 45 unique television channels that have appeared between 2004 and 2015 on the Dutch television.

4.2.3 Algorithms considered

We will now evaluate the performance of both the existing and the newly developed algorithms per experiment. For the newly developed algorithms we will consider a number of different combinations of variables.
CHAPTER 4. EXPERIMENTAL EVALUATION

For both the purely incremental algorithm and the hierarchical algorithm we will evaluate what the best depth for the breadth first search of local moves is. Furthermore we will determine whether edge flip moves have added value or if stretch and flip moves are sufficient. For the hierarchical algorithm we will furthermore determine the influence of the maximum amount of items per group.

The naming scheme to determine which combinations of parameters is currently used is as follows:

For the hierarchical incremental treemap algorithm the naming scheme will be "H(F)G[X]M[k]". The H means that the treemap is generated using the hierarchical incremental treemap algorithm. If an F is present then flip moves are considered as well. G[X] stands for the maximal amount of items that can be in a group where X denotes the amount. M[k] denotes the maximal depth considered in the breadth first search when optimizing the layout where k denotes the depth.

For the purely incremental treemap algorithm the abbreviation scheme will be "I(F)M[k]". The I means that the treemap is generated using the purely incremental treemap algorithm. If an F is present then flip moves are considered as well. M[k] denotes the maximal depth considered in the breadth first search when optimizing the layout where k denotes the depth.

The full list of algorithms that we are going to evaluate is as follows:

**Slice and Dice**  The Slice and Dice treemap algorithm [17] as presented by Schneiderman.

**Pivot-by-Middle**  The Pivot-by-Middle treemap algorithm [2] as presented by Bederson et al.


**Squarified**  The squarified treemap algorithm [5] as presented by Bruls et al. Additionally a lookahead function is implemented which is similar to the lookahead for the Strip treemap as presented by Bederson et al. [2]. This look-ahead function makes sure that the aspect ratio of the last row of the Squarified algorithm is optimized as well.

**Strip**  The Strip treemap algorithm [2] as presented by Bederson et al.

**Spiral**  The Spiral treemap algorithm [21] as presented by Tu and Shen.

**Approximation**  The Approximation treemap algorithm [15] as presented by Nagamochi and Abe.

**Moore**  The Moore treemap algorithm [20] as presented by Tak and Cockburn.

**Hilbert**  The Hilbert treemap algorithm [20] as presented by Tak and Cockburn.
4.2. PERFORMANCE OF TREEMAP ALGORITHMS

**HG7M4** The hierarchical incremental treemap algorithm. The groups contain at most 7 items and we go at most 4 local moves deep in the breadth first search when optimizing the layout.

**HG7M5** The hierarchical incremental treemap algorithm. The groups contain at most 7 items and we go at most 5 local moves deep in the breadth first search when optimizing the layout.

**HG11M4** The hierarchical incremental treemap algorithm. The groups contain at most 11 items and we go at most 4 local moves deep in the breadth first search when optimizing the layout.

**HG11M5** The hierarchical incremental treemap algorithm. The groups contain at most 11 items and we go at most 5 local moves deep in the breadth first search when optimizing the layout.

**HFG7M4** The hierarchical incremental treemap algorithm. The groups contain at most 7 items and we go at most 4 local moves deep in the breadth first search when optimizing the layout. In addition to Flip and Stretch move we will also consider edge flip moves.

**HFG7M5** The hierarchical incremental treemap algorithm. The groups contain at most 7 items and we go at most 5 local moves deep in the breadth first search when optimizing the layout. In addition to Flip and Stretch move we will also consider edge flip moves.

**HFG11M4** The hierarchical incremental treemap algorithm. The groups contain at most 11 items and we go at most 4 local moves deep in the breadth first search when optimizing the layout. In addition to Flip and Stretch move we will also consider edge flip moves.

**HFG11M5** The hierarchical incremental treemap algorithm. The groups contain at most 11 items and we go at most 5 local moves deep in the breadth first search when optimizing the layout. In addition to Flip and Stretch move we will also consider edge flip moves.

**IM3** The purely incremental treemap algorithm. We go at most 3 local moves deep in the breadth first search when optimizing the layout. This number is lower than that of the hierarchical as the number of layouts considered per depth is far larger for this algorithm. Increasing the number of moves would mean that the algorithm takes too much time. The amount of times that we will repeat the algorithm to optimize the layout will be equal to 3.

**IFM2** The purely incremental treemap algorithm. We go at most 2 local moves deep in the breadth first search when optimizing the layout. In addition to Flip and Stretch
move we will also consider edge flip moves. This number is lower than that of the purely incremental treemap algorithm without moves as considering edge flips significantly increases the number of layouts considered per depth. Increasing the number of moves would mean the algorithm takes too much time for any practical purpose. The amount of times that we will repeat the algorithm to optimize the layout will be equal to 3.

For each experiment four bar graphs will be shown. These bar charts show the influence of this experiment on the average aspect ratio, the maximum aspect ratio, the layout distance change stability score and the relative stability score. For each algorithm two data entries will be shown. The red bar on the right will show the results from the current experiment and the blue bar on the left will show the data from the baseline experiment. The aspect ratio graphs will be cut of at a aspect ratio of 20 to maintain the visibility for the small aspect ratios as the difference between 3 and 4 is far more important than the difference between 20 and 30.

Finally two scatter plot will be shown for each experiment that show the performance of the average aspect ratio and the relative stability score for each algorithm for this experiment. The newly developed algorithms will have a blue marker. The currently existing algorithms will have a red marker.

On one scatter plot we will show the performance of all algorithms and scale the axes such that all algorithms are visible. On the other scatter plot we will only show HG11M4 and IM3 from the newly developed algorithms that use local moves. The reason for this is that plotting of all the variants would make the plot illegible as the algorithms have quite similar performances. Additionally we will cut off the aspect ratio at 10 here to make sure we can focus on the most important algorithms. Moreover, the relative stability score will be cut of at 0.5 for the same reason.

### 4.2.4 Influence of the number of rectangles

From Figure 4.3 and Figure 4.4 we see that for almost all the existing algorithms the performance in terms of the stability score increase slightly when the number of rectangles in the treemap increase.

For the newly developed algorithms that utilize local moves the stability score however decrease when the number of rectangles increases. Furthermore, there does not seem to be a significant difference in the decrease between the hierarchical algorithm and the purely incremental algorithm for the stability score.

For the mean average aspect ratios of the rectangles in the treemap, the differences in values are almost negligible for all algorithms as is shown in Figure 4.6. The exception to this are the Pivot-by-X algorithms. The Pivot-by-Middle algorithm performs significantly better when the number of rectangles increase, The Pivot-by-Size algorithm has a
4.2. PERFORMANCE OF TREEMAP ALGORITHMS

Slight increase in performance and the Pivot-by-Split algorithm has a slight decrease in performance. These three algorithms however still have the highest aspect ratios except for the Spiral algorithm and the Slice and Dice algorithm.

For the mean maximum aspect ratio it is interesting that the Squarified and the Strip algorithms perform far worse when the number of rectangles increases as is shown in Figure 4.5. The Approximation algorithm on the other hand performs significantly better when the number of rectangles increases and has a extremely low mean maximal aspect ratio. For the algorithms that use local moves, we notice that the purely incremental algorithms almost double their maximum aspect ratios. This makes sense as we still allow for the same number of local moves to be performed on a larger number of total rectangles. A similar but weaker effect is displayed for most of the hierarchical algorithms. There is however one notable exception, namely the hierarchical treemap with a group size of 7 and a maximum of 4 local moves with edge flips. For this algorithm the mean maximum aspect ratio has almost tripled. It seems that the edge flips in this case were actually detrimental to the algorithm even though more layouts could be considered compared to the case where no edge flips are present. It thus seems that simply having more possible layouts to consider does not correspond directly to a lower mean maximum aspect ratio, even though it does correspond to a lower maximum aspect ratio for each timestep.

In Figure 4.7 and Figure 4.8 we see the scatter plots of the relative stability score and the mean average aspect ratio. It then becomes clear that the hierarchical algorithm is clearly the best choice for this kind of data. The incremental algorithms performs worse in terms of both stability and aspect ratio. Hilbert, Moore and Approximation treemaps perform slightly better in terms of mean average aspect ratio but are far less stable than the hierarchical algorithm. Moreover, the relative stability of the hierarchical algorithm decreases when the number of rectangles increases, while for the Hilbert, Moore and Approximation treemaps it increases. Therefore, the hierarchical algorithm becomes even more suitable when the amount of data is larger while Hilbert, Moore and Approximation algorithms only become less suited.
Figure 4.3: A bar chart showing the change in the mean relative stability score when the maximum number of rectangles in the treemap is increased from 25 to 50.

Figure 4.4: A bar chart showing the change in the mean layout distance score when the maximum number of rectangles in the treemap is increased from 25 to 50.
4.2. PERFORMANCE OF TREEMAP ALGORITHMS

Figure 4.5: A bar chart showing the change in the mean maximum aspect ratio when the maximum number of rectangles in the treemap is increased from 25 to 50.

Figure 4.6: A bar chart showing the change in the mean average aspect ratio when the maximum number of rectangles in the treemap is increased from 25 to 50.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.7: A trimmed scatter plot showing the aspect ratio and the relative stability score for each algorithm when the maximum number of rectangles in the treemap equals 50.

Figure 4.8: A complete plot showing the aspect ratio and the relative stability score for each algorithm when the maximum number of rectangles in the treemap equals 50.
4.2. PERFORMANCE OF TREEMAP ALGORITHMS

4.2.5 Influence of the size of the rectangles

In Figure 4.9 and Figure 4.10 we see that the stability score for almost all algorithms is improved significantly when the possible range of values is increased from \(1 - 100\) to \(1 - 1000\). While this might seem counter intuitive, we must keep in mind that only the range of values has increased significantly different. In particular the amount that the size of a rectangle can change per timestep has not changed. In the baseline experiment the values can change by 5% of the total range, while the value can now change by only 0.5% of the total range. Therefore, far less drastic changes will occur in the treemap which is reflected in the data. In particular we notice that the Pivot-By-X algorithms and the algorithms that use local moves have a near perfect stability score.

As we are working with data generated initially by a uniform distribution, one would expect that scaling the maximum size of the rectangles would not change the performance of the algorithms on the average aspect ratio. However, when we look at Figure 4.12 and Figure 4.11 we see that the average mean aspect ratio and average maximum aspect ratio are actually reduced for almost all of the algorithms. The reason for this again is due to how much the values can change per timestep. In the baseline experiment the sizes of the rectangles fluctuate relatively more around the initial value than in this experiment. This larger fluctuation means that the sizes of the rectangle are less close to a uniform distribution on average. The larger the differences between the sizes of the rectangles the harder it is for the algorithms to generate a treemap with good aspect ratios. If the sizes follow a uniform distribution these differences are minimized and thus better aspect ratios will be achieved.

We again notice that the presence of edge flips moves actually increases the mean maximum aspect ratio instead of decreasing it as would be expected. It thus seems that the edge flip moves optimize the maximum aspect ratio in ways that are not necessary, and are in fact even detrimental to optimizing the mean maximum aspect ratio.

In Figure 4.7 and Figure 4.8 we see the scatter plots of the relative stability score and the mean average aspect ratio. The performance of most algorithms is quite close to each other. The hierarchical algorithm performs best on the stability while achieving a low aspect ratio, but almost all other algorithms perform quite good as well.
Figure 4.9: A bar chart showing the change in the mean relative stability score when the maximum size of a rectangle is increased from 100 to 1000.

Figure 4.10: A bar chart showing the change in the mean layout distance score when the maximum size of a rectangle is increased from 100 to 1000.
4.2. PERFORMANCE OF TREemap ALGORITHMS

**Figure 4.11:** A bar chart showing the change in the mean maximum aspect ratio when the maximum size of a rectangle is increased from 100 to 1000.

**Figure 4.12:** A bar chart showing the change in the mean average aspect ratio when the maximum size of a rectangle is increased from 100 to 1000.

Stability of treemap algorithms
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.13: A trimmed scatter plot showing the aspect ratio and the relative stability score for each algorithm when the maximum size of a rectangle equals 1000.

Figure 4.14: A complete scatter plot showing the aspect ratio and the relative stability score for each algorithm when the maximum size of a rectangle equals 1000.
4.2. PERFORMANCE OF TREEMAP ALGORITHMS

4.2.6 Influence of the stability of the sizes

From Figure 4.15 and Figure 4.16 we see that for all the algorithms the performance in terms of the stability score is increased by almost a factor 2 when the amount that a rectangle can change its size increases from 5 to 25. As almost all algorithms change the layout to improve the aspect ratio it is to be expected that such large increases occur. Most algorithms have become completely unstable and would be unusable in practice if this data occurred. If we look at the algorithms that use local moves we notice that all the stability scores are around 0.2 which is still lower than even the baseline stability scores of most other algorithms. The Pivot-by-Middle and Strip algorithms still have a reasonable low stability score even though the score has increase quite a bit compared to the baseline score.

For the mean average and maximal aspect ratios we see an increase in the values for all algorithms in Figure 4.17 and Figure 4.18. The reason for this is that initially the sizes of the rectangles roughly followed a uniform distribution. When increasing the amount the sizes can change per timestep, the sizes will be less closely distributed from a uniform distribution than before. This results in large difference between the sizes of the rectangles which makes it harder to generate treemaps with good aspect ratios.

The algorithms that use local moves seem to only have a minimal increase in the performance on the aspect ratio, which is especially interesting for the purely incremental algorithms. The incremental algorithms can only make a limited number of changes to the treemap and intuitively it should thus not be able to handle fast changes in the sizes of the rectangles. However, this does not seem to be the case. It seems that with a very limited depth for the breadth first search of local moves, it is already possible to maintain a very low mean average and mean maximum aspect ratio. On the other hand the hierarchical algorithms seem to be unable to perform well on the mean maximum aspect ratio using a depth of 4 unless they are using edge flip moves as well. The depth for the breadth first search of local moves thus surprisingly seems to matter more for the hierarchical algorithms than for the purely incremental algorithms. The reason for this is that in the hierarchical algorithms the rectangle with the maximal aspect ratio is used in only a very limited amount of moves due to the compartmentalization. As it is used in a more limited amount of moves compared to the purely incremental algorithms, more moves are required before it is optimized compared to the purely incremental algorithm. Increasing the group size slightly in this case does not completely solve the problem, as there are not always enough rectangles in the treemap to utilize the increased group size efficiently.

In Figure 4.7 and Figure 4.8 we see the scatter plots of the relative stability score and the mean average aspect ratio. Most algorithms have either a stability score or a mean average aspect ratio higher than the cut-off point. Only the Pivot-by-Middle, the Strip and the algorithms using local moves are still on the plot. It seems that for data where the sizes of the rectangles change fast, one of the algorithms that uses local moves is the best algorithm to use. The Split algorithm should be considered as well as it not that far
behind in terms of performance and it calculates a treemap far faster than the algorithms that use local moves.

Figure 4.15: A bar chart showing the change in the mean relative stability score when the change in size per timestep is increased from 5 to 25.

Figure 4.16: A bar chart showing the change in the mean layout distance score when the change in size per timestep is increased from 5 to 25.
4.2. PERFORMANCE OF TREEMAP ALGORITHMS

Figure 4.17: A bar chart showing the change in the mean maximum aspect ratio when the change in size per timestep is increased from 5 to 25.

Figure 4.18: A bar chart showing the change in the mean average aspect ratio when the change in size per timestep is increased from 5 to 25.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.19: A trimmed scatter plot showing the aspect ratio and the relative stability score for each algorithm when the change in size per timestep equals 25.

Figure 4.20: A complete scatter plot showing the aspect ratio and the relative stability score for each algorithm when the change in size per timestep equals 25.
4.2.7 Influence of additions and removals

From Figure 4.15 and Figure 4.16 we see that having additions and removals of rectangles from the treemap decreases the performance of all the algorithms in terms of the stability score. The performance hit however varies drastically per algorithm. Pivot-by-Middle and Pivot-by-Split both suffer such an enormous performance hit in terms of stability. Instead of being nearly the best in terms of stability they are now almost the worst in stability. Algorithms that were already unstable on the other hand seem to only suffer a slight decrease in stability performance. For the algorithms that use local moves the stability score almost doubles but is still almost half of the stability scores of the other algorithms. These algorithms thus seem to be able to handle appearances and disappearances relatively well in terms of stability.

For the mean average and maximal aspect ratios we see small increases and decreases for the algorithms that do not use local moves as shown in Figure 4.23 and Figure 4.24. This is to be expected as these algorithms regenerate the treemap from scratch after every iteration. For the hierarchical algorithms we however see that there is a jump in the mean average aspect ratio, and a large peak in the mean maximum aspect ratio for a group size of 7. In these cases enough rectangles have been deleted from a group in the tree structure, such that the aspect ratio of the group can no longer be fixed. This shows that using hierarchical approach has some limits as well. The group size for the hierarchical approach must be large enough such that it can handle deletions. However, increasing the group size comes at the cost of increased processing power which depending on the system and the requirements might not be always feasible.

In Figure 4.7 and Figure 4.8 we see the scatter plots of the relative stability score and the mean average aspect ratio. Most algorithms have a very large stability stability score apart from the algorithms that use local moves. These algorithms and in specific the purely local moves algorithm seem to be a lot more robust in terms of stability to additions and removals of rectangles to the treemap.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.21: A bar chart showing the change in the mean relative stability score when rectangles are added and removed from the treemap.

Figure 4.22: A bar chart showing the change in the mean layout distance score when rectangles are added and removed from the treemap.
4.2. PERFORMANCE OF TREEMAP ALGORITHMS

Figure 4.23: A bar chart showing the change in the mean maximum aspect ratio when rectangles are added and removed from the treemap.

Figure 4.24: A bar chart showing the change in the mean average aspect ratio when rectangles are added and removed from the treemap.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.25: A trimmed scatter plot showing the aspect ratio and the relative stability score for each algorithm when rectangles are added and removed from the treemap.

Figure 4.26: A complete scatter plot showing the aspect ratio and the relative stability score for each algorithm when rectangles are added and removed from the treemap.

66 Stability of treemap algorithms
4.2. PERFORMANCE OF TREEMAP ALGORITHMS

4.2.8 Influence of the distribution of sizes

From Figure 4.27 and Figure 4.28 we see that having additions and removals of rectangles from the treemap increases the performance of almost all the algorithms in terms of the stability score. The reason for this is that using exponential changes in the sizes, the relative sizes of the rectangles cannot change as much as when absolute changes are used.

An interesting case is the relative small decrease in stability performance of the purely incremental layout with edge flips. As this algorithm can perform less local moves than the algorithm with a larger depth and without edge flips, it seems that the edge flips change the layout more drastically in comparison to the other local moves.

For the mean average aspect ratios and mean maximum aspect ratios quite a lot of interesting changes occur. For the hierarchical algorithms it does not seem to matter whether the distribution is log-normal or uniform for the aspect ratio. For the purely incremental algorithms we see an increase in the mean maximum aspect ratio and a slight increase in the mean average aspect ratio. The algorithms have a bit more trouble optimizing the maximum aspect ratio when the data is distributed according to a log-normal distribution. However, for most of the rectangles it will still be able to keep the aspect ratio low.

For the Pivot-by-X algorithms and the Approximation algorithm the performance on the aspect ratios improve significantly. These algorithms apparently work better under a heavy-tailed distribution such as the log-normal distribution than under a uniform distribution. For the Approximation algorithm this is not surprising. The Approximation algorithm can guarantee an aspect ratio bound depending on the ratio of the sizes of the rectangles. As in a heavy-tailed distribution the ratios are more evenly spread than in a uniform distribution it can obtain better aspect ratios. The Pivot-by-X algorithms use a similar approach as the Approximation algorithm, and the increase in performance is therefore also most likely from having a more even spread of the ratios.

The Spiral, Moore and especially the Hilbert treemaps suffer from a massive performance hit in terms of aspect ratio when a log-normal distribution is used instead of a uniform distribution. Using the log-normal random walk, a single rectangle will eventually have a far greater size than all the other rectangles. These algorithms seem to be unable to handle this as they want to lay out all the items according to the order in the data. For this ordering to work the sizes of the rectangles should however not be too far apart, or a small number of rectangles must share a side with the large rectangle which is not always the case. This thus results in large increase in the aspect ratios for these algorithms.

In Figure 4.31 and Figure 4.32 we see the scatter plots of the relative stability and the mean average aspect ratio. For the log-normal distribution a large number of algorithms perform quite nicely. Depending on whether stability or the aspect ratio is the most important either the purely incremental algorithm, the hierarchical algorithm of the Approximation algorithm is however the best suited for this kind of data. A large amount of algorithms are however almost as good and should be considered as well.
Figure 4.27: A bar chart showing the change in the mean relative stability score when the distribution is changed from a uniform distribution to a log-normal distribution.

Figure 4.28: A bar chart showing the change in the mean layout distance score when the distribution is changed from a uniform distribution to a log-normal distribution.
4.2. PERFORMANCE OF TREEMAP ALGORITHMS

Figure 4.29: A bar chart showing the change in the mean maximum aspect ratio when the
distribution is changed from a uniform distribution to a log-normal distribu-
tion.

Figure 4.30: A bar chart showing the change in the mean average aspect ratio when the
distribution is changed from a uniform distribution to a log-normal distribu-
tion.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.31: A trimmed scatter plot showing the aspect ratio and the relative stability score for each algorithm when the distribution is a log-normal distribution.

Figure 4.32: A complete scatter plot showing the aspect ratio and the relative stability score for each algorithm when the distribution is a log-normal distribution.
4.2. PERFORMANCE OF TREEMAP ALGORITHMS

4.2.9 Popular names

For the popular names data set we have generated a scatter plot showing the trade-off between the mean average aspect ratio and the relative stability score which is shown in Figure 4.33 and Figure 4.34. The purely incremental algorithm is not shown as it is unable to generate a treemap in a reasonable amount of time for this dataset due to the treemap having 200 rectangles at a time. As is shown most algorithms are able to generate treemaps with a very low aspect ratio, with the Squarified treemap algorithm achieving a mean average aspect ratio of almost 1. However, only the hierarchical algorithm is able to generate stable treemaps. All the current treemap algorithms with the exception of the Slice and dice algorithm have a relative stability score of almost 0.4 or higher. This is mostly due to the quantity of rectangles and the high addition and removal rate. The Slice and Dice algorithm is completely stable but generates treemaps with a far to large aspect ratio to be practical. The hierarchical algorithm is however able to generate extremely stable treemaps with a relative stability score of 0.05.

For the popular names dataset the hierarchical algorithm thus performs extremely well compared to the other algorithms with most of the benefits being in terms of stability.
CHAPTER 4. EXPERIMENTAL EVALUATION

Figure 4.33: A trimmed scatter plot showing the aspect ratio and the relative stability score for each algorithm on the popular names dataset.

Figure 4.34: A complete scatter plot showing the aspect ratio and the relative stability score for each algorithm on the popular names dataset.
4.2. PERFORMANCE OF TREEMAP ALGORITHMS

4.2.10 Audience ratings

For the audience ratings data set we have generated a scatter plot showing the trade-off between the mean average aspect ratio and the relative stability score which is shown in Figure 4.35 and Figure 4.36. As the audience ratings data set is significantly smaller than the popular names data set, we are able to generate a treemap using the purely incremental algorithm. We have in addition to the hierarchical algorithm with a group size of 11 and a move depth of 4 also shown the hierarchical algorithm which additionally uses edge flips as well. The reason for this is that this algorithm performs significantly better on this dataset.

Similarly to the popular names dataset all existing algorithms are unable to generate treemaps with a good performance on the relative stability indicator in the audience ratings dataset. The hierarchical algorithms and the purely incremental algorithm perform significantly better in this respect than the current existing algorithm. The reason that the existing algorithms perform badly is due to the majority of the rectangles being significantly smaller than the average. This majority of the rectangles furthermore have high fluctuations in their sizes, and furthermore there is a high addition rate which results in the instability in the existing algorithm.

The Pivot-by-X, the Moore and the hierarchical algorithm with a group size of 11 all have relatively high aspect ratios compared to the minimal aspect ratios of the other algorithms. However, the aspect ratios are still not extremely high. These high aspect ratios are due to these algorithms being unable to handle the large amount of small items well. What is quite interesting is that the hierarchical algorithm with a group size of 7, a depth of 4 and which uses edge flips performs significantly better than the hierarchical algorithm with a group size of 11 and a depth of 4. This is mostly due to decrease in the size of the groups and partially due to the presence of edge flip moves. By having smaller groups the algorithm is better able handle the large possible differences in sizes as it is easier to generate non-sliceable layouts. The edge flips additionally help reaching these non-sliceable layouts faster.

The purely incremental algorithms performs the best out of all algorithms having both one of the lowest aspect ratios and one of the lowest stability scores. Even though the incremental algorithm can not generate non-sliceable layouts easily, it is still able to handle the large amount of small items. It is able to handle the large amount of small items since it can put these together in the layout. This is in contrast to the hierarchical algorithm where the groups are fixed in the group structure.
Figure 4.35: A trimmed scatter plot showing the aspect ratio and the relative stability score for each algorithm on the audience ratings dataset.

Figure 4.36: A complete scatter plot showing the aspect ratio and the relative stability score for each algorithm on the audience ratings dataset.
4.2.11 Evaluation of the new algorithms

The hierarchical incremental moves algorithm with a group size of 11 and the incremental moves algorithm are able to maintain a high relative stability score and a high layout distance change score for all experiments. Moreover, they are also able to have a low mean maximal aspect ratio and a mean average aspect ratio. For each experiment these algorithms were either the best candidate or belonged to the best candidates for the experiment.

In general the performance of the incremental algorithm and the hierarchical algorithm is almost equal. However, the purely incremental algorithms take a significant amount of time to calculate a layout once the number of rectangles increases above 25 rectangles. When the number of rectangles becomes larger than 50 it even becomes completely unusable for on-line visualizations. The hierarchical algorithm in contrast does not suffer from this problem due to the compartmentalization of the rectangles in a hierarchical structure. The hierarchical algorithm would thus be the preferred algorithm to use until a significant speed-up can be gained in the purely incremental algorithm.

Increasing the group size of the hierarchical algorithm has a mostly positive effect on the stability performance of the hierarchical algorithms. Specifically it was able to handle additions and deletions far better with a larger group size. However, as was shown on the audience ratings dataset increasing the group size when there are large discrepancies between the sizes can increase the aspect ratio significantly. Setting the group size correctly to match the type of data encountered in the dataset visualized is thus important to generate good treemaps using the hierarchical algorithm. Having a too small group size can result in large aspect ratios when the addition and deletion rate is high. Having a too large group size can result in large aspect ratios when there is a large amount of small data appearing. Moreover, a larger group sizes means that more processing power must be used. From the experiments a group size of 7 seems to perform quite nicely in most cases and should suffice as a default value.

The edge flips moves did not seem to have a significant positive effect on the performance on the stability score and the aspect ratio measures for the hierarchical algorithms. In fact it has a negative impact on the performance for a number of experiments. For the incremental algorithms it is be better to be able to traverse a level deeper into breadth first search using local moves than to use edge flips. Whether increasing the depth the algorithm is able to traverse any further has any significant effect remains to be tested. This is however a very time-consuming process as the incremental algorithm is currently not fast enough to be able to handle a larger maximum depth.

For the hierarchical algorithm it seems that increasing the maximal depth the algorithm is able to traverse rarely has a significant positive effect. It thus seems that having a depth of 4 is already sufficient from group sizes of 7 and 11. If the group sizes are increased to higher values we expect that the depth should be increased as well to obtain similar
CHAPTER 4. EXPERIMENTAL EVALUATION

results, but this has not been tested yet.

4.2.12 Evaluation of the relative stability score

From the results of the experiment one more very interesting observation can be made. The distribution of relative stability scores for the algorithms is almost always quite similar to the distribution of the layout distance change stability scores disregarding scaling. While the layout distance change function score and the relative stability score calculate different aspects of the stability score one would expect the scores to differ quite drastically. However, this does not seem to be the case. This indicates that the relative stability and the layout distance change function partially encode the same information.

The experiments we performed were however not aimed to determine the relation between the relative stability score and the layout distance change functions score. We can therefore not draw any conclusions on the nature of the relation between the two stability measures from the experiments directly. It would be interesting to determine how exactly these two stability measures obtain such similar distributions, but due to time constraints this is not included in this thesis.
Chapter 5

Conclusions

In this thesis we have explored the concept of stability in rectangular treemap algorithms. In Section 2 we have briefly evaluated a number of contributing factors of stability and evaluated how the existing definitions of stability take these components into account. We found that none of the current definitions take the change in group structure in the treemap into account. In response we have developed a new stability measure using the relative position of the rectangles in the treemap that does encode the change in group structures in the treemap.

In Section 3 we have developed the concept of local moves. Local moves can be used to transform an existing treemap into a different treemap using small incremental changes. We have proven that using only Flip and Stretch local moves we are able to transform any treemap into any other treemap. Furthermore, we have used the concept of local moves to develop two new treemap algorithms. These two new algorithms are the first treemaps algorithms that are able to generate non-sliceable rectangular treemaps.

In Section 4 we have evaluated the newly developed treemap algorithms. The newly developed treemap algorithms were able to maintain a low mean average aspect ratio, a low mean maximum aspect ratio, a low layout distance change functions score and a low relative stability score. In all experiments the newly developed algorithms were among the best compared to all current existing rectangular treemap algorithms in terms of both the mean average aspect ratio and the stability score. Furthermore they were are able to achieve a good balance between two quality indicators which no other algorithm was able to do consistently.
CHAPTER 5. CONCLUSIONS

5.1 Open problems

A number of problems remain unsolved in this thesis.

The first open problem is that a user study need to be performed to verify that the change in group structures and the relative stability are indeed important for the perceived stability of the treemap.

The second open problem is to determine a complete definition of stability that includes all contributing factors of stability. When such a definition is developed a conclusive verdict can be given on the stability of each treemap algorithm. Moreover it would become possible to adapt the search method of the local moves to only consider the moves that are the most stable. This would result in the algorithm becoming even more stable.

A third open problem is to determine the relationship between the layout distance function score and the relative stability score. While these two stability measures work using a completely different basis to determine a stability score, the resulting scores seem to follow roughly the same distribution. It would be interesting to see exactly why these two measures give such similar distribution. We believe that this information would help significantly in setting up a complete definition of stability.

The fourth open problem is to determine an upper bound on the average case maximal aspect ratio of the hierarchical incremental algorithm. We believe that it should be possible to give an upper bound on the average maximal aspect ratio given that a distribution is used that has a finite mean.

Finally it should be investigated whether there exists a strongly polynomial time algorithm to convert a layout to a treemap for which the sizes of the rectangles are correct. If such an algorithm can be found it will dramatically speed up the newly developed algorithms. This would mean that the algorithms can handle a larger number of rectangles and can also consider more possible layouts.
Bibliography


Appendix A

Properties of rectangular layouts

In this appendix we will give a number of proofs that concern the properties of rectangular layouts.

A.1 Types of rectangular layouts

Rectangular layouts can be divided into two types of layouts which each have their own properties, namely sliceable layouts and non-sliceable layouts.

A.1.1 Sliceable rectangular layouts

**Lemma 5.** Let rectangle $A$ denote a rectangle in a sliceable rectangular layout $L$ where the number of rectangles in $L$ is at least 2. Then there must exist a one-sided inner maximal segment $ms$ such that rectangle $A$ is the only rectangle adjacent to one side of $ms$. We denote such a rectangle as a grounded rectangle.

We prove Lemma 5 lemma by contradiction. Let rectangle $A$ denote a rectangle in the treemap. We assume that there exists a sliceable rectangular layout $L$, where there does not exist an inner maximal segment $ms$ such that rectangle $A$ is the only rectangle adjacent to one side of $ms$. In other words, no side of the rectangle associated to rectangle $A$ is in itself an inner maximal segment.

Rectangle $A$ can then be in six different positions:

- **Position 1** It is adjacent to exactly zero sides of the bounding box.
- **Position 2** It is adjacent to exactly one side of the bounding box.
APPENDIX A. PROPERTIES OF RECTANGULAR LAYOUTS

Position 3 It is adjacent to exactly two opposite sides of the bounding box.
Position 4 It is adjacent to exactly two orthogonal of the bounding box.
Position 5 It is adjacent to exactly three sides of the bounding box.
Position 6 It is adjacent to exactly four sides of the bounding box.

We now derive a contradiction for each of the possible positions of rectangle A.

Position 1:

Let $ms_T$ be the maximal segment adjacent to rectangle A. Without loss of generality we assume that rectangle C is above rectangle A. $ms_T$ cannot be completely contained in the top segment of rectangle A, as it would otherwise hold that the top segment of rectangle A is a maximal segment in itself. It must thus hold that $ms_T$ is either longer to the left, longer to the right or longer on both sides as the top segment of rectangle A.

We first consider the case that the $ms_T$ end further to the left and to the right than the top segment of rectangle A. This results in the layout as shown in Figure A.1.

![Figure A.1: $ms_T$ ends further to the left and to the right than the top segment of rectangle A.](image)

Let $ms_L$ be the maximal segment adjacent to the left of rectangle A. Let $ms_R$ be the maximal segment adjacent to the right of rectangle A. If $ms_L$ and/or $ms_R$ end above $ms_T$, then it would mean that $ms_L$ and/or $ms_R$ would intersect the maximal segment $ms_T$ resulting in a degenerate case. To solve this degenerate case we would either have to split $ms_T$ into multiple maximal segments or we would need to split $ms_L$ and $ms_R$ in multiple maximal segments. If we split $ms_T$ into multiple segments, then $ms_T$ would not longer end further to the left and right than the top segment of rectangle A which results in a contradiction. If we split $ms_L$ and/or $ms_R$ in multiple maximal segment $ms_L$ and/or $ms_R$ would no longer end further to the top than the top segment of rectangle A which again results in a contradiction. Thus $ms_L$ and $ms_R$ must both end at $ms_T$.

As the left and right segment of rectangle A cannot be maximal segment by themselves, it must hold that $ms_L$ and $ms_R$ end below the left and right segment of rectangle A. This results in the layout as shown in Figure A.2.
Finally let $ms_B$ be the maximal segment adjacent to the bottom of rectangle A. It $ms_B$ ends to the left of $ms_L$ or to the right of $ms_R$ we have a degenerate case again. To solve this degenerate case we would have to split either $ms_L$ or $ms_R$ into multiple maximal segments. If we would split up one of these maximal segments $ms_L$ and $ms_R$, it would no longer hold that both of these maximal segments end below the left/right segment of rectangle A which results in a contradiction. Therefore, $ms_B$ must lie between $ms_L$ and $ms_R$. This means that $ms_B$ is completely contained in the bottom segment of rectangle A which results in a contradiction.

We now consider the case that the maximal segment $ms_T$ ends further to one side top side of rectangle A. Without loss of generality we assume that it is ends to the right side of rectangle A as is shown in Figure A.3.

Let $ms_R$ be the maximal segment adjacent to the right of rectangle A. If $ms_R$ ends above the top segment of rectangle A, then we again have a degenerate case where we would have to split either $ms_T$ or $ms_R$ into multiple maximal segments. In both cases a contradiction can be derived using the same reasoning as for the previous cases. $ms_R$ must thus end below the bottom segment of rectangle A. This results in the layout as is shown in Figure A.4.
Let $ms_B$ be the maximal segment adjacent to the bottom of rectangle $A$. If $ms_B$ ends to the right of the right segment of rectangle $A$, we again have a degenerate case where we would have to split either $ms_R$ or $ms_B$ into multiple maximal segments. In both cases a contradiction can be derived using the same reasoning as for the previous cases. $ms_B$ must thus end to the left of the left segment of rectangle $A$. This results in the layout as is shown in Figure A.5.

Let $ms_L$ be the maximal segment adjacent to the left of rectangle $A$. If $ms_L$ ends below the bottom segment of rectangle $A$, we again have a degenerate case where we would have to split either $ms_B$ or $ms_L$ into multiple maximal segments. In both cases a contradiction can be derived using the same reasoning as for the previous cases. $ms_L$ must thus end above the top segment of rectangle $A$ as is shown in Figure A.6. It is not possible to slice the layout in two over any of the maximal segments $ms_T, ms_R, ms_B, ms_L$ as long as the other segments are still present. The resulting layout is therefore a non-sliceable layout. We had assumed that the resulting layout was sliceable and thus we have derived a contradiction. As $ms_C$ can not end further to the right, further to the left, or further on both sides than the top segment of rectangle $A$ we have derived a contradiction for this position.
A.1. TYPES OF RECTANGULAR LAYOUTS

Figure A.7: Rectangle A touches the top and bottom side.

Figure A.6: \( m_{SL} \) ends above the top segment of rectangle A.

Position 2:
Without loss of generality we assume that rectangle A is adjacent to the top side of the bounding box. This case is then identical to position 1, where the maximal segment \( m_{ST} \) equals the top side of the bounding box. Therefore, this position also leads to a contradiction.

Position 3:
Without loss of generality we assume that rectangle A is adjacent to the top and the bottom side of the bounding box as is shown in Figure A.7. Let \( m_{SR} \) be adjacent to the right of rectangle A. As rectangle A touches both the top and the bottom side of the bounding box it is not possible for \( m_{SR} \) to end above or below the top/bottom segment of rectangle A. \( m_{SR} \) must thus be contained in the right side of rectangle A which gives us a contradiction for this position.

Position 4:
Without loss of generality we assume that rectangle A is adjacent to the top and right sides of the bounding box. This case is then identical to position 1, where the maximal segment \( m_{ST} \) equals the top side of the bounding box and the maximal segment \( m_{SR} \) equals the right side of the bounding box. Therefore, this position also leads to a contradiction.

Stability of treemap algorithms 85
APPENDIX A. PROPERTIES OF RECTANGULAR LAYOUTS

Position 5:
Without loss of generality we assume that rectangle A is adjacent to the top, right and bottom sides of the bounding box. This case is then identical to position 1 where the maximal segment $ms_T$ equals the top side of the bounding box, the maximal segment $ms_R$ equals the right side of the bounding box and the maximal segment $ms_B$ equals the bottom side of the bounding box. Therefore, this position also leads to a contradiction.

Position 6:
If rectangle A touches all four boundaries, it must be the only rectangle in the layout. As the layout $L$ should have at least 2 rectangle this position leads to a contradiction.
As all of the possible positions of rectangle A lead to a contradiction, we have proven that Lemma 5 must hold.

A.1.2 non-sliceable rectangular layouts

Lemma 6. Let rectangle A be a rectangle that is not grounded. Rectangle A must then be adjacent to 4 inner maximal segments.

To prove Lemma 7 we will assume that rectangle A is not adjacent to 4 inner maximal segments. Rectangle A must then be adjacent to at least one of the boundary segments. Without loss of generality we assume it rectangle A is adjacent to the left boundary segment. Let $ms_T$ be the maximal segment adjacent to the top of rectangle A. Furthermore let $ms_B$ be the maximal segment adjacent to the bottom of rectangle A and let $ms_R$ be the maximal segment adjacent to the right of rectangle A.
As rectangle A is adjacent to the left boundary and rectangle A is not grounded, it must hold that both $ms_T$ and $ms_B$ end to the right of the top/bottom side of rectangle A. This is shown in Figure A.8.

![Figure A.8](image.png)

Figure A.8: rectangle A touches the left boundary and $ms_T$ and $ms_B$ both end to the right of the top/bottom segment of rectangle A.
As rectangle A is not grounded it must hold that $ms_R$ ends below the bottom segment of rectangle A and/or above the top segment of rectangle A. If $ms_R$ ends above the top segment of rectangle A an intersection would occur with $ms_T$ resulting in a degenerate case. To solve this degenerate case we would either have to split $ms_R$ into multiple maximal segments or we would need to split $ms_T$ into multiple maximal segments. If we split $ms_T$, then $ms_T$ would not longer end further to the right than the top segment of rectangle A which results in a contradiction. If we split $ms_R$ in multiple maximal segments, then $ms_R$ would no longer end above the top segment of rectangle A which again results in a contradiction. Thus the top of $ms_R$ must end at $ms_T$.

If $ms_R$ ends below the bottom segment of rectangle A an intersection would occur with $ms_B$ resulting in a degenerate case. Using the same reasoning as before this leads to a contradiction again. Therefore, the bottom of $ms_R$ must end at $ms_B$.

This means that the right side of rectangle A must be a maximal segment and thus rectangle A is a one-sided rectangle and we have derived a contradiction proving Lemma 6.

**Lemma 7.** Let rectangle A be a rectangle that is not grounded. Rectangle A must then be adjacent to the endpoints of 4 inner maximal segments.

From Lemma 6 we know that rectangle A must be adjacent to 4 inner maximal segments. We thus only need to prove that rectangle A must be adjacent to the endpoints of these inner maximal segments.

We assume that rectangle A is not adjacent to at least one endpoint of an inner maximal segment. Without loss of generality we assume that rectangle A is not adjacent to the endpoint of the maximal segment $ms_l$ to the left of rectangle A. Then $ms_L$ must end below the bottom segment of rectangle A and above the top segment of rectangle A.

Let $ms_T$ be the maximal segment that is adjacent to the top of rectangle A. Furthermore let $ms_B$ be the maximal segment that is adjacent to the bottom of rectangle A and let $ms_R$ be the maximal segment that is adjacent to the right of rectangle A. If $ms_T$ ends to the left of the left segment of rectangle A an intersection would occur with $ms_L$ resulting in a degenerate case.

To solve this degenerate case we would either have to split $ms_T$ into multiple maximal segments or we would need to split $ms_L$ into multiple maximal segments. If we split $ms_T$, then $ms_T$ would not longer end further to the left than the left segment of rectangle A which results us a contradiction. If we split $ms_L$ in multiple maximal segments, then $ms_L$ would no longer end above the top segment of rectangle A which again results in a contradiction. Thus, as rectangle A is not grounded, $ms_T$ must end to the right of rectangle A. Similarly $ms_B$ must end to the right of rectangle A. The resulting layout is shown in Figure A.9.
As rectangle A is not grounded it must hold that \( ms_R \) ends below the bottom segment of rectangle A and/or above the top segment of rectangle A. If \( ms_R \) ends above the top segment of rectangle A an intersection would occur with \( ms_T \) resulting in a degenerate case. Using the same reasoning as before this leads to a contradiction again. Thus the top of \( ms_R \) must end at \( ms_T \).

If \( ms_R \) ends below the bottom segment of rectangle A an intersection would occur with \( ms_B \) resulting in a degenerate case. Using the same reasoning as before this leads to a contradiction again. Therefore, the bottom of \( ms_R \) must end at \( ms_B \).

This means that the right side of rectangle A must be a maximal segment. Rectangle A is thus a grounded rectangle and we have derived a contradiction proving Lemma 7.
Appendix B

Lower bounds on minimal maximum aspect ratio of rectangular layouts

B.1 Aspect ratios of sliceable and non-sliceable rectangular layouts

This appendix serves the purpose of proving that for sliceable layout the optimal aspect ratio can be tightly lower bounded. The lower bound equals $\sqrt{\frac{s(B)}{s(A)}}$ where rectangle A is the rectangle with the smallest size and rectangle B is the rectangle with the second smallest size. Moreover we will prove that non-sliceable layouts can obtain better maximal aspect ratios than this lower bound.

B.1.1 Lower bound on the aspect ratio of sliceable rectangular layouts

We will use Lemma 5 to lower bound the optimal maximal aspect ratio. Let rectangle A be the rectangle with the smallest weight in a sliceable rectangular layout $L$. Then let B be a rectangle that is adjacent to rectangle A over a maximal segment $ms$, such that rectangle A is the only rectangle on one side of $ms$. The existence of such a rectangle B follows from Lemma 5. Without loss of generality we assume that rectangle B is adjacent to the left side of rectangle A. The right side of rectangle B is thus completely contained in the left side of rectangle A. It then holds that $h(B) \leq h(A)$. As $s(B) \geq s(A)$ we furthermore know that $w(B) \geq w(A)$.

We now prove a lowerbound on the optimal maximal aspect ratio by distinguishing four possible cases:
APPENDIX B. LOWER BOUNDS ON MINIMAL MAXIMUM ASPECT RATIO OF
RECTANGULAR LAYOUTS

Case 1: \( h(A) > w(A) \land h(B) > w(B) \)
Case 2: \( h(A) > w(A) \land h(B) \leq w(B) \)
Case 3: \( h(A) \leq w(A) \land h(B) > w(B) \)
Case 4: \( h(A) \leq w(A) \land h(B) \leq w(B) \)

Case 1

In case 1 it holds that \( h(A) > w(A) \land h(B) > w(B) \) which is depicted in Figure B.1.

![Figure B.1: h(A) > w(A) \land h(B) > w(B)](image)

As \( h(A) > w(A) \) and \( h(B) > w(B) \) the aspect ratios of rectangles A and B are respectively \( \frac{h(A)}{w(A)} \) and \( \frac{h(B)}{w(B)} \). A lower bound on the optimal aspect ratio can thus be given as:

\[
\max \left( \frac{h(A)}{w(A)}, \frac{h(B)}{w(B)} \right).
\]

As \( h(B) \leq h(A) \) and \( w(B) \geq w(A) \) it holds that \( \frac{h(A)}{w(A)} \) is always larger than \( \frac{h(B)}{w(B)} \). We can thus rewrite this bound to \( \frac{h(A)}{w(A)} \). We now rewrite this lower bound in terms of \( s(A) \) and \( s(B) \).

\[
\frac{h(A)}{w(A)} \geq \frac{h(B)}{w(A)} = \frac{s(B) \cdot h(A)}{s(A) \cdot w(B)} \geq \frac{s(B) \cdot \sqrt{s(A)}}{s(A) \cdot \sqrt{s(B)}} = \sqrt{\frac{s(B)}{s(A)}}
\]

We thus have a lower bound on the minimal maximum aspect ratio of \( \sqrt{\frac{s(B)}{s(A)}} \) for this case.

Case 2

In case 2 it holds that \( h(A) > w(A) \land h(B) \leq w(B) \).
As \( h(B) \leq w(B) \), the aspect ratio of rectangle B is minimized if \( h(B) = h(A) \) regardless of the choice of \( w(A) \) and \( w(B) \) which is depicted in Figure B.2. We will therefore assume that \( h(B) = h(A) \) for the remainder of this case.

![Figure B.2: \( h(A) = h(B) > w(A) \wedge h(B) \leq w(B) \)]

We define \( x \) to be equal to \( s(B) / s(A) \). We now rewrite \( w(B) \) in terms of \( x \) and \( w(A) \).

\[
s(B) = x \cdot s(A) \implies w(B) \cdot h(B) = x \cdot w(A) \cdot h(A) \\
\implies w(B) \cdot h(B) = x \cdot w(A) \cdot h(B) \\
\implies w(B) = x \cdot w(A)
\]

The aspect ratios of rectangles A and B can now be represented as \( \frac{h(A)}{w(A)} \) and \( \frac{w(A) \cdot x}{h(A)} \) respectively. The maximum aspect ratio of rectangles A and B can never be lower than the average aspect ratio of rectangles A and B. Therefore, the average aspect ratio of rectangles A and B is also a lower bound on the minimal maximum aspect ratio:

\[
\text{avg} = \frac{\frac{h(A)}{w(A)} + \frac{w(A) \cdot x}{h(A)}}{2}
\]

We will now find the value of \( w(A) \) that minimizes the average. The value of \( h(A) \) will follow from the choice of \( w(A) \). We will find this value by computing the partial derivative of \( \text{avg} \) with regard to \( w(A) \), and computing when this partial derivative equals 0.

\[
\frac{\partial}{\partial w(A)} \left( \frac{\frac{h(A)}{w(A)} + \frac{w(A) \cdot x}{h(A)}}{2} \right) = 0 \implies \frac{x}{2h(A)} = \frac{h(A)}{2w(A)^2} = 0 \\
\implies \frac{x}{2h(A)} = \frac{h(A)}{2w(A)^2} \\
\implies w(A)^2 = \frac{h(A)^2}{x} \\
\implies w(A) = \frac{h(A)}{\sqrt{x}} \wedge w(A) = -\frac{h(A)}{\sqrt{x}} \\
\implies w(A) = \frac{h(A)}{\sqrt{x}} \ (h(A), x \text{ and } w(A) \text{ are all positive}).
\]
Filling in the found value of \( w(A) \) then gives the lower bound on the minimal maximum aspect ratio:

\[
\text{avg} = \frac{\frac{h(A)}{w(A)} + \frac{w(A) \times x}{h(A)}}{2}
\]

\[
= \frac{\frac{h(A)}{\sqrt{x}} + \frac{h(A)}{h(A)}}{2}
\]

\[
= \frac{2}{\sqrt{x}}
\]

\[
= \sqrt{x}
\]

\[
= \sqrt{s(B)}
\]

\[
= \sqrt{s(A)}
\]

**Case 3**

In case 3 it holds that \( h(A) \leq w(A) \) and \( h(B) > w(B) \). Furthermore it holds that \( h(B) \leq h(A) \leq \sqrt{s(A)} \leq \sqrt{s(B)} \). However, as \( h(B) > w(B) \), it must hold that \( h(B) > \sqrt{s(B)} \) giving rise to a contradiction. Therefore, this case is not possible.

**Case 4**

In case 4 it holds that \( h(A) \leq w(A) \wedge h(B) \leq w(B) \) which is depicted in Figure B.3.

![Figure B.3: h(A) ≤ w(A) ∧ h(B) ≤ w(B)](image)

As \( h(A) \leq w(A) \) and \( h(B) \leq w(B) \) the aspect ratios of rectangles A and B are respectively \( \frac{w(A)}{h(A)} \) and \( \frac{w(B)}{h(B)} \). A lower bound on the optimal aspect ratio can thus be given as \( \max \left( \frac{w(A)}{h(A)}, \frac{w(B)}{h(B)} \right) \). As \( h(A) \geq h(B) \) and \( w(A) \leq w(B) \), it holds that \( \frac{w(B)}{h(B)} \) is always larger than \( \frac{w(A)}{h(A)} \). We can thus rewrite the bound to \( \frac{w(B)}{h(B)} \). We now rewrite this lower bound in terms of \( s(A) \) and \( s(B) \):
B.1. ASPECT RATIOS OF SLICEABLE AND NON-SLICEABLE RECTANGULAR LAYOUTS

\[
\frac{w(B)}{h(B)} \geq \frac{w(B)}{h(A)} = \frac{s(B)w(A)}{s(A)h(B)} \geq \frac{s(B)\sqrt{s(A)}}{s(A)\sqrt{s(B)}} = \sqrt{\frac{s(B)}{s(A)}}
\]

**Lower bound**

Combining the lower bounds found in the case distinction we obtain a lower bound for any sliceable rectangular layout \( L \) that contains more than 1 rectangle:

\[
\min \left( \sqrt{\frac{s(B)}{s(A)}}, \sqrt{\frac{s(B)}{s(A)}}, \sqrt{\frac{s(B)}{s(A)}} \right) = \sqrt{\frac{s(B)}{s(A)}}
\]

A is here the rectangle with the smallest size and B is the rectangle with the second smallest size.

**B.1.2 The lower bound is tight**

We proof that this lower bound is tight using the layout shown in Figure B.4. In Figure B.4 rectangles \( B, C, D, E \) all have a size of 8 while rectangle \( A \) has a size of \( \epsilon \). \( \epsilon \) will be picked such that the aspect ratio equals \( \sqrt{\frac{s(B)}{s(A)}} \).

The width and height of the bounding box both equal \( \sqrt{32 + \epsilon} \). The width of rectangle \( A \) must then be equal to \( \frac{8 + \epsilon}{\sqrt{32 + \epsilon}} \) in the layout. The height of rectangle \( A \) then becomes \( \epsilon \frac{\sqrt{32 + \epsilon}}{8 + \epsilon} \).

The aspect ratio of rectangle \( A \) then becomes the maximum of \( \frac{\frac{8 + \epsilon}{\sqrt{32 + \epsilon}}}{\frac{\epsilon \sqrt{32 + \epsilon}}{8 + \epsilon}} = \frac{(8 + \epsilon)^2}{\epsilon(32 + \epsilon)} \) and \( \frac{\epsilon \sqrt{32 + \epsilon}}{8 + \epsilon} = \frac{\epsilon(32 + \epsilon)}{(8 + \epsilon)^2} \). We now pick \( \epsilon \) such that \( \sqrt{\frac{8}{\epsilon}} = \frac{(8 + \epsilon)^2}{\epsilon(32 + \epsilon)} \).
\[
\sqrt{\frac{8}{\epsilon}} = \frac{(8 + \epsilon)^2}{\epsilon(32 + \epsilon)} \implies \\
(32 + \epsilon)\sqrt{\frac{8}{\epsilon}} = (8 + \epsilon)^2 \implies \\
\frac{(32 + \epsilon)2\sqrt{2}}{\sqrt{\frac{1}{\epsilon}}} = (8 + \epsilon)^2 \implies \\
2\sqrt{2}\epsilon + 64\sqrt{2} = (8 + \epsilon)^2 \frac{1}{\sqrt{\epsilon}} \implies \\
2\sqrt{2}\epsilon + 64\sqrt{2} = (64 + 16\epsilon + \epsilon^2) \frac{1}{\sqrt{\epsilon}} \implies \\
(2\sqrt{2}\epsilon + 64\sqrt{2})^2 = (64 + 16\epsilon + \epsilon^2)^2 \implies \\
\epsilon(2\sqrt{2}\epsilon + 64\sqrt{2})^2 = (64 + 16\epsilon + \epsilon^2)^2 \implies \\
8\epsilon^3 + 512\epsilon^2 + 8192\epsilon = \epsilon^4 + 32\epsilon^3 + 384\epsilon^2 + 2048\epsilon + 4096 \implies \\
\epsilon^4 + 24\epsilon^3 − 128\epsilon^2 − 6144\epsilon + 4096 = 0 \implies \\
\]

which is a quadratic equation in standard form. The real solutions to this equation are:

\[
\epsilon = -6 + 6\sqrt{5} - 2\sqrt{6\sqrt{5} - 2} \approx 0.659 \\
\epsilon = 2(-3 + 3\sqrt{5} + \sqrt{6\sqrt{5} - 2} \approx 14.174
\]

As the size of rectangle A needed to be smaller than the size of rectangle B we pick \(\epsilon\) to be equal to \(-6 + 6\sqrt{5} - 2\sqrt{6\sqrt{5} - 2}\). As the aspect ratio of rectangle \(A\) equals \(\sqrt{\frac{s(B)}{s(A)}}\) the lower bound is tight.

Calculating the aspect ratio gives us a bound of approximately 3.48.

### B.1.3 non-sliceable layouts can outperform optimal sliceable layouts

The lower bound of \(\sqrt{\frac{s(B)}{s(A)}}\) does not hold for non-sliceable layouts. Using the same rectangle weights as in Figure B.4 we can give an non-sliceable layout that achieves an aspect ratio that is smaller than 3.47. This layout is shown in Figure B.5 where rectangle A is now not a grounded rectangle. The aspect ratio here is bounded by \(\frac{\sqrt{2}}{\epsilon}\) which equals:
Figure B.4: Rectangle A has a size of $\epsilon$, which gives a tight lower bound on the aspect ratio for $\epsilon = -6 + 6\sqrt{5} - 2\sqrt{6\sqrt{5} - 2}$.

$$y = \frac{1}{2} \left( \frac{\sqrt{32 + \epsilon} + \sqrt{\epsilon}}{\sqrt{32 + \epsilon} - \sqrt{\epsilon}} \right) = \frac{\sqrt{32 + \epsilon} + \sqrt{\epsilon}}{\sqrt{32 + \epsilon} - \sqrt{\epsilon}} = \frac{32 + \epsilon - \epsilon}{32 + \epsilon - \epsilon} = \frac{32 + 2\epsilon + 2\sqrt{32 + \epsilon + \epsilon^2}}{32} = \frac{32}{16 + \epsilon + \sqrt{32\epsilon + \epsilon^2}} = 1 + \frac{\epsilon}{16} + \frac{\sqrt{32\epsilon + \epsilon^2}}{16}$$

Filling in $\epsilon = -6 + 6\sqrt{5} - 2\sqrt{6\sqrt{5} - 2}$ gives us an aspect ratio of $1 + \frac{\epsilon}{16} + \frac{\sqrt{32\epsilon + \epsilon^2}}{16} < 1.3311$. This is smaller than the the lower bound of $\approx 3.48$ for the sliceable layout. Therefore, the lower bound of $\sqrt{\frac{b(B)}{b(A)}}$ does not hold for non-sliceable layouts. Non-sliceable layouts can thus outperform sliceable layouts.
Figure B.5: The aspect ratio is bounded by \( \frac{1}{2} (\sqrt{32 + \epsilon} + \sqrt{\epsilon}) \) and \( \frac{1}{2} (\sqrt{32 + \epsilon} - \sqrt{\epsilon}) \).