MASTER’S THESIS

Visualizing Business Information using Generalized Treemaps

by

Roel Vliegen

Supervisors:

Prof. dr. ir. Jack van Wijk (TU/e)
Dr. Erik-Jan van der Linden (MagnaView B.V.)

Eindhoven, March 2006
Abstract

More and more data is collected everywhere. In corporate environments, information is collected and used to support decision making. Often business graphics are used to show aggregations of the collected business information.

Treemaps, a visualization method, are well-known in the visualization community, but not readily available for exploring business information. Treemaps provide both detail and overview in a single image.

In this thesis we show how to visualize business information using treemaps. Firstly, we present a method for generating tree structures from tabular business information. Next, we propose generalized treemaps, which enable the creation of many new variations on treemaps and business graphics. These variations combine strong points of both treemaps and business graphics.

Additionally, we have developed MagnaView, a commercial visualization tool that implements all proposed techniques. We present a selection of variations on business graphics and treemaps that MagnaView can create.
Contents

0. Preface ......................................................................................................................... 1
1. Introduction .................................................................................................................. 2
  1.1. Assumptions and Requirements ............................................................................. 3
  1.2. Overview ............................................................................................................... 4
2. Related Work ............................................................................................................... 5
  2.1. Business Graphics ............................................................................................... 5
  2.2. Tree Visualization ............................................................................................... 6
     2.2.1. Node-Link Diagrams .................................................................................... 7
     2.2.2. Treemaps ..................................................................................................... 8
     2.2.3. Icicle Plots ................................................................................................. 12
  2.3. Evaluation ......................................................................................................... 13
  2.4. Conclusion ........................................................................................................ 15
3. Data ........................................................................................................................... 16
  3.1. Relations ........................................................................................................... 16
  3.2. Generating Tree Structures ............................................................................... 17
  3.3. Implementation ................................................................................................. 18
     3.3.1. Only One Relation .................................................................................... 18
     3.3.2. Unique Values ........................................................................................... 18
     3.3.3. Expressions and Maps ............................................................................... 18
     3.3.4. Late Data Binding ..................................................................................... 20
     3.3.5. Tree Generator .......................................................................................... 20
4. Generalizing Treemaps ............................................................................................. 22
  4.1. Base Treemap Algorithms ................................................................................ 22
  4.2. Properties of treemap algorithms ...................................................................... 24
     4.2.1. Definitions ................................................................................................. 24
     4.2.2. Properties .................................................................................................. 25
  4.3. Generalized Treemap Algorithm ...................................................................... 26
5. Extending Treemaps ................................................................................................. 31
  5.1. Level Based Layouts ......................................................................................... 31
  5.2. Treemaps and Business Graphics ..................................................................... 33
     5.2.1. Size Method .............................................................................................. 33
     5.2.2. Sort Method .............................................................................................. 36
     5.2.3. Direction Method ...................................................................................... 37
     5.2.4. Desired Aspect Ratio ................................................................................ 38
     5.2.5. Vanishing Point ......................................................................................... 40
     5.2.6. Transformations ......................................................................................... 42
     5.2.7. Margins ..................................................................................................... 46
     5.2.8. Adjust for Uniform Density ........................................................................ 47
     5.2.9. Draw Line to Parent .................................................................................. 49
     5.2.10. Matrix Layouts .......................................................................................... 50
     5.2.11. Empty Nodes ............................................................................................. 52
6. Results ....................................................................................................................... 53
  6.1. Requirements .................................................................................................... 53
  6.2. Analysis ............................................................................................................. 54
6.2.1. Records Level ........................................................................................................ 55
6.2.2. Structural Levels .................................................................................................. 56
6.2.3. Layout Levels ...................................................................................................... 64
6.3. Real World Examples ............................................................................................ 65
   6.3.1. Cases of a Notary’s Office .................................................................................. 65
   6.3.2. Grades of a High-School .................................................................................. 67
   6.3.3. XML Structured Text ........................................................................................ 71
6.4. Rules of Thumb ...................................................................................................... 72
7. Conclusion .................................................................................................................. 74
   7.1. How to Visualize Business Information Using Treemaps ................................ 74
   7.2. MagnaView ......................................................................................................... 75
   7.3. Recommendations for Future Research .............................................................. 76
   7.4. Possible Improvements on MagnaView .............................................................. 78
References ....................................................................................................................... 80
0. Preface

This master’s thesis is part of the Computer Science program of the author at the Technische Universiteit Eindhoven.

The Visualization Group, part of the Department of Mathematics and Computer Science of the Technische Universiteit Eindhoven, directed by Jack van Wijk, created SequoiaView, a visualization tool which visualizes the content of a hard disk and can be used to answer the question “Why is my hard disk full?”.

SequoiaView has an XML interface for storing snapshots of the files and directories in a file system. Erik-Jan van der Linden, at the time consultant in the area of, amongst others, management information, used this XML interface to visualize business information using SequoiaView. This proved to be useful and Erik-Jan van der Linden and Jack van Wijk decided to explore the possibilities of using treemaps for visualizing business information using an internship, which I fulfilled.

During my internship, which lasted from April till July 2004, I implemented a treemap solution that visualizes trees extracted from business information. The results looked promising and it was decided to start MagnaView B.V. to commercialize the product. MagnaView B.V. is a joint venture between Tilia Cordata IT B.V., Technische Universiteit Eindhoven and myself, founded in July 2005 and located in the MultiMediaPaviljoen on the TU/e Campus.

After a second internship at Philips Electronics Nederland, I continued work on this master’s thesis in January 2005.

As part of the master’s thesis MagnaView Viewer and MagnaView Designer are developed at MagnaView B.V. under my responsibility. Both are already used commercially. In this report MagnaView is used both as the company name and as a designator for the products.

I want to express my special gratitude to Jack for his valuable remarks and Erik-Jan for his vision which made this all possible.

I also want to thank Alwin, Anita, Dennis, Guido, Huub, Jake, Marc, Rogier, Sefer, Thomas, Tomasz and Willem for their support and work at MagnaView.

Finally, I wish to express my gratitude to my parents and sister who have always supported me during my education.

Roel Vliegen
1. Introduction

More and more data is collected everywhere. In corporate environments, business information is collected and used to support decision making. Tables or listings are used to inspect the information on a detailed level. Aggregates are calculated to make trends and global patterns visible.

Visualization uses the visual system of users to quickly transfer large amounts of information in a structured way. This enables analysis of larger data sets than tables and aggregation alone permit. Visualization, especially when using so-called pre-attentive elements, can move the burden of reasoning partially to the visual system, freeing other parts of the cognitive system for higher-level tasks. This can lead to immediate insight in complex phenomena that otherwise would be hard to grasp.

Besides this, visualization can also provide context to search results and make outliers visible. Context and outliers are present, but often not perceivable in tables; aggregations often condense information to single facts, leaving out context and outliers altogether.

In practice, the value of visualization is recognized, resulting in the frequent use of business graphics like bar and pie charts. Although suitable for presenting a couple of values into a single image, detailed information is still omitted.

Treemaps are a visualization technique that can simultaneously display both detail and overview of tree structures. Treemaps have been used successfully for visualizing various kinds of data, such as the content of file systems [1, 2], stock market data [3], process control data [4], and source code of large programs [5, 6]. However, treemaps are still not often used for visualizing business information.

This leads to the central question of this thesis:

*How to visualize business information using treemaps?*

In this thesis we give an answer to this question. In search for this answer, we encountered many new and exciting possibilities which raised additional questions. Some of these are answered in this thesis, others will hopefully be addressed in future work. All methods and techniques have been implemented in a system, called MagnaView.
1.1. Assumptions and Requirements

In order to understand the role of MagnaView as an application we start with a simple model. The model is an adaptation of a model presented by Wilson and Bergeron [7]. It shows the relations between MagnaView and the environment and the way a user interacts with both, i.e.,

![Diagram of MagnaView model]

The adequate use of information in a business environment puts a number of requirements on visualization [8, 9, 10, 43, 44, 46]. For this research project, we have defined the following assumptions and requirements.

First of all, MagnaView should enable corporate users to better understand their business. Visualization is not the primary aim; nevertheless, an assumption here is that it can be a useful tool for making well informed decisions.

Second, MagnaView should effortlessly fit in the environment of a corporate user. This implies that MagnaView must be able to:
- run on hardware available in a corporate environment; a reasonably up to date configuration is assumed regarding processor and memory capabilities, however, no particular capabilities of the video hardware are assumed,
- run on the main operating systems available in corporate environments, i.e., Microsoft Windows 2000 or higher,
- read data from miscellaneous data sources, such as text files, Microsoft Office files and databases,
- integrate in information systems using seamless two way interoperability, i.e., the information system should be able to start and load data in MagnaView and MagnaView should be able to initiate actions in the information system.
Third, MagnaView should present information to users which at least comprises the level of detail which is part of the everyday routine of the professional in question. Teachers, for example, are interested in individual students; aggregates, such as average mark per class, are often too abstract. Notaries know what kinds of cases they deal with. Often these professionals even have a vivid mental image of specific entities; teachers know the faces of students, notaries clearly remember particular cases. To accommodate this level of detail, MagnaView should use treemaps for visualizing business information. Business information is assumed to be available in tables. Treemaps, on the other hand, visualize explicit tree structures. Therefore, MagnaView must be able to transform tabular business information into tree structures. This is further detailed in Chapter 3. Chapter 4 contains more information about treemaps.

Fourth, MagnaView should allow users to use common business graphics like pie and bar charts. Common business graphics are suitable for expressing aggregations or overviews of data, but specific entities are lost. MagnaView must unify business graphics and treemaps to generate visualizations that combine the strong points of both business graphics and treemaps, thus:

- presenting users with visualizations they are accustomed to, while still benefiting from the strong points of treemaps,
- utilizing the past experience of users with standard business graphics, resulting in more accessible visualizations and faster learning curves,
- providing users with a stepping stone for a transition to more advanced visualizations.

In Chapter 5 extensions to treemaps are presented that enable the creation of treemaps that have the look and feel of business graphics.

Fifth, MagnaView should allow explorative data analysis to users. Not all tree structures are best represented using the same visualization method, therefore, users must be enabled to interactively adjust both the mapping from tabular data to tree structures as well as the visual representation. This way the visualization evolves together with the data analysis and benefits the users during the complete analysis process. We call the combinations of data mappings and visualization settings views. The settings that make up views are implicitly explained in Chapters 3, 4 and 5. In Chapter 6 we present some rules of thumb for creating views and some examples of real world views.

1.2. Overview

Chapter 2 of this thesis first presents an overview of various related visualization methods and implementations, next, an informal evaluation shows both strong and weak points of the presented methods. In Chapter 3 we present the data model that is used in MagnaView and how tabular data is transformed into tree structures. We present an introduction to treemaps in Chapter 4, followed by a detailed description of our generalized treemap algorithm. In Chapter 5 we discuss various extensions to this algorithm which enable the creation of many different variations on treemaps and business graphics. We present a selection of those variations in Chapter 6. Finally, in Chapter 7 we present some conclusions and future work.
2. Related Work

In this chapter we present an overview of related work using real world examples and implementations. For each example or implementation we summarize the methods and techniques that are used. We have not opted for a categorization of methods and techniques, because we are interested in the limitations of each implementation, which result from the chosen combination of techniques and methods.

In Section 2.1 a short overview of business graphics is presented. Next, in Section 2.2 we present several tree visualization implementations and examples. Section 2.3 shows some differences and relations between the examples and implementations presented. Finally, Section 2.4 contains some concluding remarks.

2.1. Business Graphics

Business graphics, like bar and pie charts, are widely used. Many variations exist and most people take them for granted because they have become used to them. Nearly everybody understands business graphics, they are even taught in primary schools. This is advantageous, because users do not need to learn how to use business graphics anymore and can start using them right away.

The power of business graphics lies in their simplicity. Essentially, areas are used to represent quantities. So, comparing quantities becomes equal to comparing areas, thus enabling the comparison of many quantities in one glimpse.

An early, but quite sophisticated, example of this is Minard’s Tableau Graphique, created December 1845 [11],

![Minard's Tableau Graphique](image)

which uses independently both the horizontal and vertical axes, respectively for distance and quantity of goods transported by ship. The product of distance and quantity equals
cost, which becomes visible as the area of a rectangle. Color is used to represent the type of goods; labels are used to annotate the image.

From the early work of Minard to modern business graphics, many variations have been created [12, 13, 14]. However, surprisingly little has changed in the past 160 years.

At the moment, *Microsoft Excel* [15] is the most widely used tool for creating Business Graphics like

![Stacked columns](image1)

![Clustered columns](image2)

![100% stacked columns](image3)

![Pie charts](image4)

Remarkably, these charts created using Excel are all direct descendants of works of Minard. Only the presentation differs; the principles remain unchanged.

Stacked columns are useful for comparing totals, while clustered columns are better suited for comparing the individual components. Ratios between components can be made visible using 100% stacked columns and pie charts are useful for presenting ratios between totals.

### 2.2. Tree Visualization

Tree visualization is an area of information visualization concerned with representing tree structures. In tree visualization, the input data consists of an explicit tree structure; optionally attributes for nodes and links can be present. The content of this chapter is based on an extensive study of the literature on visualization of tree structures. Although our study may not be exhaustive, the overview below presents a good introduction to the various factors on which visualizations of tree structures differ.
2.2.1. Node-Link Diagrams

Node-Link Diagrams are the most straightforward representations of tree structures. They are often used when explaining tree structures to students, because both the nodes and the links of the tree structure can be drawn explicitly, like, for instance

![Node-link diagram](image)

Node-link diagrams are very suitable for representing small trees and for showing high-level structure. For larger trees, node-link diagrams quickly become cluttered to the point that individual nodes or links can no longer be distinguished [16, 17].

*Dendrograms* [18] use horizontal and vertical lines instead of diagonal lines for representing links. Besides that, often nodes are not represented explicitly, but implicitly as branches of the links, resulting in images like

![Dendrogram](image)
Radial node-link layouts [19, 20] use the available space more efficiently by using a radial layout algorithm which maps the inner nodes to the center and all leaf nodes to the complete periphery, like, for instance,

MeSH dataset using radial tree layout.

The periphery is $\pi$ times longer than the minimum of the width and height of the display area. In case of near square display areas, like when using computer displays, valuable space is gained where it is most needed. Animation is used to make transitions more clear.

2.2.2. Treemaps

Treemaps map tree structures to rectangles. The root node of a tree structure is mapped to the initial rectangle where the treemap has to be shown. Next, this rectangle is divided into several smaller rectangles: one for each child node of the root node. This process is repeated recursively for each new rectangle until all rectangles represent leaves. The relations of a tree structure are represented using inclusion and not using links like node-link diagrams do. Chapter 4 contains a more detailed description of treemaps.

Optionally a size can be assigned to each leaf node. In this case, the treemap algorithm generates rectangles which area is in proportion to the size of the corresponding leaf nodes. This makes treemaps suitable for displaying quantitative attributes of leaf nodes.
Treemaps are strongly related to *mosaic displays*, which are used in statistical graphics and show values in a contingency table cross-classified by one or more factors [21], like, for instance

![mosaic of the relation between hair color and eye color.](image)

Area represents the *cell frequency*, i.e., the frequency of a combination of factors. The width and height of the cells are fractions of the factors, i.e. hair color and eye color. Mosaic displays can be generated using the traditional treemap algorithm.

*TreeViz*, created by Shneiderman and Johnson, is the original implementation of treemaps. It uses the slice-and-dice algorithm [1], which we call the traditional treemap algorithm in this thesis, for creating layouts of the files and directories on a hard disk. The color of a rectangle is based on the type of the associated file, the size of a rectangle on the size of a file of directory, resulting in visualizations like

![treemap created using TreeViz.](image)
Many elements that are used in newer treemap implementations are already present in the original implementation. Borders, for example, can be added around nodes to enhance the hierarchical structure, resulting in nested treemaps like

![Nested Treemap](image)

Zooming enables exploration of data in more detail and more detailed information is shown for nodes selected using the mouse cursor. Interesting is the use of vertical labels, consisting of individual horizontal characters.

**SequoiaView** [2] also visualizes the files and directories on hard disks. It uses cushion treemaps [22], which use shading to enhance the hierarchical structure, and squarified treemaps [23], to generate layouts that have better aspect ratios. This result in images like

![Cushion Treemap](image)

which have the added benefit of looking attractive. Cushion treemaps effectively use 2.5D elements [24], i.e., 3D cues added to a 2D layout. Besides zooming, SequoiaView also has the option to display more or less detail. More detailed information of the selected file is presented in a tool tip.
PhotoMesa [25] is an application for browsing pictures. It combines two new layout algorithms, called strip treemaps [26] and quantum treemaps [26] into one visualization, resulting in

![PhotoMesa application.](image1)

Interesting is that the visualization is non space-filling in the strictest sense: the overall structure is space-filling, but the pictures do not fill the categories completely. This is the result of a custom size function for inner nodes that only uses sizes that are multiples of a single input dimension. The strip treemap is implemented using a look-ahead function that eliminates final skinny strips.

PhotoMesa uses both a treeview and treemaps, thus acknowledging that they have different uses; one is not always better than the other. Zooming is supported by a visual zooming cue. The selected image is shown enlarged in a tool tip.

Beamtrees [27], also based on treemaps, make inner nodes visible by using a non space-filling approach [28]. This adds occlusion as a depth cue, i.e.,

![Beamtrees.](image2)

The 3D view enables users to use a suitable view point for a task. Nodes can be colored by clicking on them.
2.2.3. Icicle Plots

Icicle plots [29] are used as statistical graphics for representing the results of hierarchical clustering. Icicle plots subdivide one axis to represent the tree structure. We show in Section 5.2 that this has similarities to inclusion as used in treemaps. Icicle plots represent inner nodes explicitly, this in contrary to treemaps (without borders) where inner nodes are completely covered by leaf nodes. Icicle plots display all nodes of a tree simultaneously just like node-link diagrams. We now show the same tree structure, using a node-link diagram, an icicle plot and a nested treemap [1], respectively

In Chapter 5 we present a framework in which we unify these representations.

SunBurst [30] uses a radial layout, but is otherwise quite similar to Icicle plots [29], i.e.,

Three navigation methods are explored in SunBurst, namely angular detail, detail outside and detail inside [31]. Each method combines an overview with a more detailed view. Smooth animations are used to change between two states.
InterRing [32] uses the same principle as SunBurst, i.e.,

Many user interaction options are provided, i.e., besides drill-down, zooming and panning also rotation and distortion are added. Distortion [39] is used as a solution for the zoom/context problem, creating a “multi-focus, intuitive, space efficient distortion process” [33]. On top of this, several selection, highlighting and coloring options are offered.

2.3. Evaluation

In this section we first present some observations on the examples of the previous sections. Next, we present a subjective assessment of all methods presented.

- **Finite area** is a limitation that nearly all visualization techniques struggle with. The number of pixels is always limited so it must be decided what to use them best for. Especially the choice between representing quantity and expressing structure using area must be addressed. Each technique balances the usage of area in a different way, leading to different strong and weak points.

Treemaps use the same area for both representing quantity and expressing structure, but this does not lead to optimally perceivable structure [29], i.e., it is often not immediately clear which nodes are siblings or what the depth of a node in the tree structure is.
- Relations in tree structures can either be represented using links or inclusion. Links make a tree structure clearly visible and are easy to understand. For small tree structures links are preferable, but for larger tree structures links lead to clutter [34].

We also observe that treemaps use inclusion effectively in two dimensions, which is contrary to icicle plots that use inclusion only in one dimension [35]. Using one dimension leads to a more clearly perceivable structure, but also limits the number of relations that can be shown.

- Inclusion relations can be made clearer by using borders or shading [22]. Shading has the advantage of not using valuable area, but puts some limitations on the further use of color.

One completely unrelated advantage of shading is that it looks attractive, which is important, because user satisfaction is a factor leading to the success of visualization systems [8, 36].

- The appearance of the various techniques is strongly affected by the choice of coordinate system. Cartesian layouts are most suitable if area is used for expressing quantities. Radial layouts, like pie charts, can also be used for this, but are less precise and more error prone [37, 38]. On the other hand, radial layouts are more suitable for displaying ratios and offer layout benefits for representing structure [31]. Furthermore, users like using radial layouts.

- Interaction can be used to partially counter the problem of limited available area by adding or enhancing relevant information when needed. Overlays add detail or structural information on mouse movements. Zooming can enlarge areas a user is interested in. Distortion acts much like zooming, but also keeps the context visible. Selecting the level of detail enables a user to choose between detail and overview. Multiple view points allows for individual inspection of different aspects of the data without having to combine all information in one view.

All visualization methods have their strong and weak points. We, subjectively, assessed each method on four criteria, which we derived from the user requirements.
1. Structure indicates the suitability for expressing hierarchical relations that are created from factors (attributes) that users want to examine.
2. Detail indicates the suitability for expressing detailed information for individual data points. Users not only request overviews, they also want to inspect individual elements and use them as examples for making the overviews tangible.
3. Quantitative indicates the suitability for expressing quantitative information, i.e., both aggregates and values of individual nodes, which is important to visualize quantitative information like profits or hours of work.
4. *Scalability* indicates an estimate of the number of nodes that can be reasonably represented using the method. Users have the disposal of more and more information, so visualization methods should be scalable.

For each criterion an assessment is presented, where “++” indicates that the method performs very well on the criterion and “--” indicates that the method does not perform well on the criterion. If a criterion is not applicable or neutral it is left blank. Criteria that are unclear are marked with “?”. Scalability is expressed as the number of nodes that can reasonably be displayed using the method. The methods are sorted on scalability. The conflicting relation between the criteria is clearly visible, i.e.,

<table>
<thead>
<tr>
<th>Visualization Method</th>
<th>Structure</th>
<th>Detail</th>
<th>Quantitative</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node-Link Diagram</td>
<td>++</td>
<td>-</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>Mosaic Display</td>
<td>+</td>
<td>-</td>
<td>++</td>
<td>100</td>
</tr>
<tr>
<td>Dendrogram</td>
<td>++</td>
<td>-</td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>Icicle Plot</td>
<td>++</td>
<td>+</td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>Radial Dendrogram</td>
<td>++</td>
<td>-</td>
<td></td>
<td>3000</td>
</tr>
<tr>
<td>SunBurst</td>
<td>++</td>
<td>+</td>
<td></td>
<td>3000</td>
</tr>
<tr>
<td>InterRing</td>
<td>++</td>
<td>+</td>
<td></td>
<td>3000</td>
</tr>
<tr>
<td>Traditional Treemap</td>
<td>+</td>
<td>?</td>
<td>+</td>
<td>50000</td>
</tr>
<tr>
<td>Beam Tree</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>50000</td>
</tr>
<tr>
<td>Squarified Treemap</td>
<td>--</td>
<td>+</td>
<td>++</td>
<td>100000</td>
</tr>
<tr>
<td>Strip Treemap</td>
<td>-</td>
<td>+</td>
<td>++</td>
<td>100000</td>
</tr>
<tr>
<td>Bar Chart</td>
<td>--</td>
<td>++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pie Chart</td>
<td>--</td>
<td>++</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.4. Conclusion

A subjective assessment of the visualization methods presented was chosen, because objective measurements are hard to obtain. Small differences in implementation or additional features can skew the results and favor particular methods for the wrong reasons. This again does show the importance of rigid user tests. Unfortunately few suitable user tests are available in our discipline.

None of the methods excels in all aspects. The assessment shows that the desired properties conflict with each other. Depending on the questions asked and datasets used, different strategies must be pursued to obtain the best results. Our goal is to combine several methods to obtain visualizations that are suitable for displaying both structure and detail and can be used for presenting quantitative information, while remaining scalable.
3. Data

Business information is mostly stored in relational databases or simple tables and is characterized by a tabular layout. As such, the data has no explicit hierarchical structure, but often implicit hierarchical structures are present in the data. These implicit structures can be made explicit by imposing a hierarchical organization of the data [7, 40].

MagnaView reads tabular data from miscellaneous data sources, such as text files, Microsoft Office files and databases, and converts the data into tree structures.

We first discuss the internal data model in Section 3.1. In Section 3.2 the generation of trees is presented. Finally, in Section 3.3, we present some implementation details.

3.1. Relations

First, we extend the model of Section 1.1. Only the parts that are related to data handling are shown, i.e.,

![Diagram of data model](image)

An important requirement is that a variety of data sources can be read and processed, such as text files, XML files, Microsoft Office files and databases (ODBC). Furthermore, multiple distinct tree structures can be derived from the same input data. Therefore, first, the data is loaded into an intermediate structure, i.e., a two dimensional table.

In this table, data is represented as a relation $D$. Each column $i$ of the table is an attribute and has a name $A_i$. Each row of the table is a tuple of values; one value for each attribute. The complete relation $D$ is a set of tuples, i.e.,

$$D = \{(v_1, v_2, ..., v_N)\},$$

where $v_i$ is a value for attribute $A_i$ for a tuple.
A projection of relation $D$ on attributes $A_1, \ldots, A_n$ is denoted by

$$\pi_{A_1,\ldots,A_n}(D)$$

and denotes the relation $D$ restricted to columns for attributes $A_1, \ldots, A_n$.

Often, when creating visualizations, the values contained in the relation do not suffice and derived values are required. Therefore, new attributes can be added to the relation. Each new attribute is a function $f$ of the other attributes, i.e.,

$$A_{\text{new}} = f(A_1, \ldots, A_n).$$

### 3.2. Generating Tree Structures

Each view defines the tree structure that should be generated. For that purpose, a view contains a list of levels, i.e., $L_{i}, 1 \leq i \leq M$. Each level $L_{i}$ is associated with one attribute $B_{i} = A_{L_{i}}$, which is used to generate the tree structure.

A tree is a special kind of graph, i.e., a directed acyclic graph that consists of a set of nodes $V$ and a set of directed edges $E$.

The set of nodes $V$ is defined as

$$V = \bigcup_{0 \leq i \leq M + 1} V_{i},$$

where

$$V_{i} = \begin{cases} \{\}\quad & i = 0 \\ \pi_{B_{1},\ldots,B_{i}}(D) & 1 \leq i \leq M \\ D & i = M + 1 \end{cases}$$

This defines the nodes of a tree. It contains a node for each tuple of the relation $D$ and a root node $n_{\text{root}}$, designated by the empty tuple. Furthermore, for each level $i$, it contains a node for each unique tuple after the projection on the attributes $B_{1},\ldots,B_{i}$.

Each node $c$, except the root node $n_{\text{root}}$, has one outgoing edge $\langle c, p \rangle$. This edge denotes a parent-child relation between the child $c$ and the parent $p$.

The set of edges $E$ is defined as

$$E = \bigcup_{1 \leq i \leq M + 1} E_{i},$$

where

$$E_{i} = \{\langle c, p \rangle | c \in V_{i} \land p \in V_{i-1} \land p \in \pi_{B_{i-1},\ldots,B_{i}}(\{c\})\} \quad 1 \leq i \leq M + 1 .$$
This defines the edges of the tree such that each node \( c \), except \( n_{root} \), has an edge to a unique parent node \( p \). Thus, \( p \) is a generalization of \( c \), which groups all tuples that are similar to \( c \), i.e., all tuples \( s \) for which \( \pi_{B_1, \ldots, B_m} ([c]) = \pi_{B_1, \ldots, B_m} ([s]) \).

The depth of a node \( n \) is defined as the length of the path from \( n \) to the root node \( n_{root} \). A level contains all nodes that have equal depth.

### 3.3. Implementation

In this section we present some implementation details of MagnaView.

#### 3.3.1. Only One Relation

It was decided to use only one table (relation) in MagnaView to keep the implementation as simple as possible. This also ensures a one-on-one mapping between the input data, i.e., the tuples of the relation, and the nodes in the treemap. Nearly all data that effectively can be visualized using treemaps can be joined to a single table [38]. Although simple and proven to be useful in our case, this creates a bottleneck if larger datasets are used, because joining multiple relations often leads to duplication of information. If needed, this can be addressed in the future by implementing a more advanced data model.

#### 3.3.2. Unique Values

Often, a relation \( D \) contains many duplicate values for an attribute \( A \), i.e., \( |\pi_A(D)| \) is much smaller than \( |D| \). Therefore, MagnaView stores, for an attribute \( A \), all unique values only once in a list \( U_A \). Each tuple does not contain the actual value \( v \) of attribute \( A \), but the index of \( v \) in \( U_A \). Storing only the unique values for each attribute not only lowers the memory requirements, but more important, allows for actions to be carried out on the unique values. One of our real-world data sets contains \( |D| = 197,516 \) tuples about the grades of a high-school, but \( |\pi_{\text{studentname}}(D)| = 1427 \) and \( |\pi_{\text{grade}}(D)| = 94 \). Converting all grades from strings to doubles only leads to 94 type conversions, which results in a significant speed-up.

Storing only unique values has proven to be very useful in practice, because many (interesting) attributes tend to have few unique values, thus leading to significant performance improvements.

#### 3.3.3. Expressions and Maps

In MagnaView, a function of attributes can be defined in two ways, namely via expressions and maps. Maps are suitable for discrete ranges; expressions are suitable for continuous ranges.
Expressions are formulas that are evaluated for each tuple to create new attribute values. They are suitable for calculating numerical values, like, for instance,

\[ A_{\text{profit}} = A_{\text{revenue}} - A_{\text{cost}} \]

where \( A_{\text{revenue}} \) and \( A_{\text{cost}} \) are existing attributes and \( A_{\text{profit}} \) is a new attribute. For each tuple the value of the new attribute \( A_{\text{profit}} \) is calculated by taking the value of the attribute \( A_{\text{revenue}} \) and subtracting the value of the attribute \( A_{\text{cost}} \).

Expressions are also suitable for performing string operations, like, for instance,

\[ A_{\text{firstcharacter}} = \text{leftstr}(A_{\text{name}}, 1) \]

where the new attribute \( A_{\text{firstcharacter}} \) is defined as the left most character of attribute \( A_{\text{name}} \).

Maps are used for mapping values of attributes to categories. A category is a new value that denominates some specific kind of values for a specified attribute. Maps are suitable for grouping values, like, for instance,

\[ A_{\text{assessment}}: A_{\text{Grade}} \rightarrow \{\text{Failed, Passed, With Honors}\} \]

\[
\begin{align*}
\text{if} & \quad 0,0 < \text{Grade} \leq 5,5 \quad \rightarrow \text{Failed} \\
\text{elseif} & \quad 5,5 < \text{Grade} \leq 9,0 \quad \rightarrow \text{Passed} \\
\text{elseif} & \quad 9,0 < \text{Grade} \leq 10,0 \quad \rightarrow \text{With Honors} \\
\text{else} & \rightarrow \text{Unknown}
\end{align*}
\]

where for each tuple the value of \( A_{\text{assessment}} \) is determined by first checking whether the “Failed” category holds for the value of the \( A_{\text{Grade}} \), if the predicate does not hold, then the next category is tried and so on.

MagnaView uses three kinds of categories:
- **Value categories** contain discrete set of values. The value of the new attribute is equal to the first category that contains the value of the associated attribute, for instance,

\[ A_{\text{country}}: A_{\text{brand}} \rightarrow \{\text{France, German, Netherlands}\} \]

\[
\begin{align*}
\text{if} & \quad \text{brand} \in \{\text{Renault, Peugeot}\} \quad \rightarrow \text{France} \\
\text{elseif} & \quad \text{brand} \in \{\text{BMW, Mercedes}\} \quad \rightarrow \text{German} \\
\text{elseif} & \quad \text{brand} \in \{\text{Spyker}\} \quad \rightarrow \text{Netherlands} \\
\text{else} & \rightarrow \text{Unknown}
\end{align*}
\]

where “BMW” and “Mercedes” are both mapped to category “German”.

- **Range categories** contain a minimum value and a maximum value. The value of the new attribute is equal to the first category for which the minimum is lower than the value of the associated attribute and the maximum is higher, for instance

\[
A_{\text{age}}: A_{\text{year}} \rightarrow \{\text{Old}, \text{Young}\} \\
\text{if } 2002 < \text{year} \leq 2006 \rightarrow \text{Young} \\
\text{elseif } 1800 < \text{year} \leq 2002 \rightarrow \text{Old} \\
\text{else } \rightarrow \text{Unknown} \\
\text{fi}.
\]

- **Expression categories** contain an expression on one or more attributes. The value of the new attribute is equal to the first category for which the expression evaluates to true for the value of the associated attribute(s), for instance,

\[
A_{\text{buy}}: A_{\text{age}} \times A_{\text{country}} \rightarrow \{\text{buy, do_not_buy}\} \\
\text{if } (\text{age} = \text{Young}) \text{ and } (\text{country}=\text{German}) \rightarrow \text{buy} \\
\text{elseif } (\text{country} = \text{Netherlands}) \rightarrow \text{buy} \\
\text{else } \rightarrow \text{do_not_buy} \\
\text{fi}.
\]

All three categories can be used together in a single map. Furthermore each map can use a category which collects all tuples for which no other category holds, i.e., the `else` clauses in the examples.

### 3.3.4. Late Data Binding
Most management information tools extract data from an information system, load this data and transform the data (ELT, where L is Load). The result is stored in proprietary format. For MagnaView it was decided to always load data from the original data source and not store the loaded data in a proprietary format together with the other project settings. If the data changes in the original data source, then these changes will also reflect in the visualizations that MagnaView creates. This allows for better integration, because the user is freed from data conversion steps. Data conversion is a barrier that must be broken down, otherwise users are reluctant to use a tool [8]. Note that MagnaView still allows the use of data stored in proprietary or intermediate format.

### 3.3.5. Tree Generator
MagnaView stores the tree structure as an object tree of `TreemapNodes` that are implemented as

```
TreemapNode = record
  parent : TreemapNode;
  children : list of TreemapNode;
  ...
end;
```
For ease of use and performance reasons, both the parent and children of each node are stored. To avoid further duplication of information about the tree structure, other attributes of the nodes, such as the color and size values, are also stored in the TreemapNode data structure.

MagnaView builds the tree incrementally. First, the root node $n_{root}$ is created. Next, for each tuple $t$ of relation $D$ a node $n_t$ is created. Node $n_t$ tries to traverse the path $\langle \rangle, \pi_{B_1}(t), \ldots, \pi_{B_{\ldots}\ldots B_n}(t), t$ in the tree. If an edge of the path is not yet present, then the edge and destination node are added.
4. Generalizing Treemaps

In this chapter we first summarize three base treemap algorithms, namely traditional treemaps, squarified treemaps and strip treemaps. Next, we consider some properties of these treemap algorithms, and finally we define a generalized treemap algorithm which can be parameterized into the base treemap algorithms and some variations hereof.

4.1. Base Treemap Algorithms

This section provides a short overview of the traditional treemap, squarified treemap and strip treemap algorithms. All three algorithms divide an initial rectangle recursively into smaller rectangles. The difference between the algorithms lies in the layout algorithms that are used for the smaller rectangles.

The traditional treemap [1] algorithm divides the rectangle into several strips. The first child gets the first strip, the second child the second strip and so on. The areas of the strips are in proportion to the size of the child nodes. This is also called the slice-and-dice method.

Each division step, the algorithm alternates between horizontal and vertical strips. Starting with vertical strips, one and two division steps of some sample dataset lead respectively to

![Diagram showing one division step and two division steps.](image)

So the vertical strip 3 of the first division step is divided in the second division step into three horizontal strips, namely strips 3.1, 3.2 and 3.3. The aspect ratios of the rectangles tend to become large, i.e., the rectangles are often very elongated and thin. This makes it difficult for a user to estimate the size of a rectangle or to select a rectangle using a pointing device [37].

The squarified treemap algorithm [23] was designed to solve this problem. The design goal was to keep the aspect ratios of the rectangles as close to 1 as possible. The optimal solution turned out to be NP-complete and as such computationally intractable. Therefore, a heuristic method was used.
The squarified treemap algorithm first sorts the children on size in descending order. Next, just like in the traditional treemap algorithm, strips are created, only this time a strip can consist of more than one rectangle. The number of rectangles in a strip is determined by a greedy algorithm which adds rectangles to the strip as long as doing so improves the worst aspect ratio of the rectangles in the strip.

Additionally, the orientation of a strip is either horizontal or vertical, depending on which orientation is expected to provide better aspect ratios. Using the same sample data as above, the squarified treemap algorithm leads for one and two divisions steps to, respectively

![one division step](image1) ![two division steps](image2)

In the first division step two horizontal strips are created. The first strip contains the nodes 2 and 3; the second strip the nodes 1, 4 and 5. In the second division step, node 2 is divided into two vertical strips; the first consists of nodes 2.1 and 2.2, the second of node 2.3. Node 5 is divided into two horizontal strips, the first contains node 5.2, and the second contains nodes 5.3 and 5.1.

Strip treemaps [26] were created to solve a problem introduced by squarified treemaps: squarified treemaps indeed create treemaps that are close to optimal regarding the aspect ratios, however, the order of the rectangles is destroyed. Looking up a particular node can be very hard. Just imagine finding a particular word in a dictionary that is no longer alphabetically ordered.

The strip treemap algorithm squarifies the rectangles, while keeping them ordered. The algorithm is quite similar to the squarified treemap algorithm, but there are three differences:

1. the rectangles are not sorted on size, and thus order is preserved,
2. the orientation of the strips is fixed and alternating, just like in the traditional treemap algorithm,
3. the average aspect ratio of a strip is optimized, not the worst aspect ratio.
Again using the same sample data as above, the strip treemap algorithm leads for one and two division steps to, respectively

The first division step results in three vertical strips. The first strip contains nodes 1 and 2, the second nodes 3 and 4 and the last node 5. In the second division step all nodes are divided into horizontal strips. Node 1, for instance, is divided into two horizontal strips. The first contains nodes 1.1 and 1.2, the second node 1.3.

4.2. Properties of treemap algorithms

In this section, first, we present some definitions, next, we use these definitions to define four properties of treemap algorithms.

4.2.1. Definitions

Treemap algorithms map nodes of a tree structure $T=(V,E)$ to hierarchical partitions of a subset of $\mathbb{R}^2$ such that the area of each partition is proportional to the (calculated) size of the associated node. We only consider rectangular partitions. A rectangular partition $r$ is the set of all points in $\mathbb{R}^2$ bounded by $r_{\text{xmin}}, r_{\text{xmax}}, r_{\text{ymin}}, \text{ and } r_{\text{ymax}}$, i.e.,

$$r \equiv \{ (x,y) \in \mathbb{R}^2 \mid r_{\text{xmin}} \leq x < r_{\text{xmax}} \land r_{\text{ymin}} \leq y < r_{\text{ymax}} \} .$$

The area $a(r)$ of a rectangular partition $r$ is geometrically defined as

$$a(r) = (r_{\text{xmax}} - r_{\text{xmin}})(r_{\text{ymax}} - r_{\text{ymin}}) .$$

We define the aspect ratio of a rectangular partition $r$ as the smallest ratio of the width and the height, thus

$$\text{AspectRatio}(r) = \min \left\{ \frac{r_{\text{xmax}} - r_{\text{xmin}}}{r_{\text{ymax}} - r_{\text{ymin}}}, \frac{r_{\text{ymax}} - r_{\text{ymin}}}{r_{\text{xmax}} - r_{\text{xmin}}} \right\}$$

such that $0 \leq \text{AspectRatio}(r) \leq 1$.
The treemap algorithm associates each node \( n \in V \) of the tree structure with a rectangular partition \( n_{\text{rect}} \).

The size \( S(n) \) of a node \( n \) is a value of \( n \) or the sum of the sizes of the children of \( n \). The size can express a size metric, but also, for example, such properties as confidence, importance or relevance. The mapping of size is usually chosen such that more important nodes have larger sizes.

The area \( A(n) \) of a node \( n \) is defined as \( A(n) = a(n_{\text{rect}}) \), i.e., the area of the rectangular partition \( n_{\text{rect}} \) representing \( n \).

The depth \( \text{depth}(n) \) of a node \( n \) is the distance from \( n \) to the root node \( n_{\text{root}} \) as defined in Section 3.2.

Each node, except the root node \( n_{\text{root}} \), has a parent \( n_{\text{parent}} \) such that \( \langle n, n_{\text{parent}} \rangle \in E \).

The set of children \( n_{\text{children}} \) of a node \( n \) contains all nodes the parent of which is \( n \), i.e.,

\[
\{ m \mid \langle m, n \rangle \in E \}.
\]

### 4.2.2. Properties

In this section we present some properties of treemap algorithms. Although other properties are important as well, we now only focus on the properties of the layout algorithms that characterize treemaps. These properties are partly similar to the properties presented in [1].

The input to a treemap algorithm consists of the root node \( n_{\text{root}} \) of a tree structure \( T = (V, E) \) and a region \( n_{\text{root}_{\text{rect}}} \). The treemap algorithm subdivides \( n_{\text{root}_{\text{rect}}} \) into hierarchical partitions representing the tree structure. After execution of the treemap algorithm the space-filling, non-overlapping, inside parent and uniform density properties hold, which have the following meanings.

**Space Filling (SF):** One of the original design goals of the traditional treemap algorithm was not to waste any space. This in contrast to other techniques, like node-link diagrams, that use space inefficiently. Treemaps always use all available space; treemaps are so-called space filling. Strict space filling (SSF) treemaps use all space for leaf nodes, i.e.,

\[
\text{SSF} : \left( \bigcup n : n \in V \land n_{\text{children}} = \emptyset : n_{\text{rect}} \right) \supseteq n_{\text{root}_{\text{rect}}} ,
\]

while space filling (SF) treemaps can also use space for internal nodes, i.e.,

\[
\text{SF} : \left( \bigcup n : n \in V \land n \neq n_{\text{root}} : n_{\text{rect}} \right) \supseteq n_{\text{root}_{\text{rect}}} .
\]
**Non-overlapping (NO):** The second property of treemap algorithms is that for each node all of the child nodes are non-overlapping, i.e.,

\[
\forall c, d : c \in V \land d \in V \land c_{\text{parent}} = d_{\text{parent}} \land c \neq d : c_{\text{rect}} \cap d_{\text{rect}} = \emptyset.
\]

This guarantees that all, sufficiently large, nodes are visible; no data is occluded.

**Inside Parent (IP):** To spatially preserve the parent-child relationship of the tree structure each node is located within its parent, i.e.,

\[
\forall n : n \in V \land n \neq n_{\text{root}} : n_{\text{rect}} \subseteq n_{\text{parent}\text{rect}},
\]

which implies that the region of a parent bounds the regions of its children. This also enables spatial search algorithms that use bounding boxes to quickly eliminate all children of nodes.

**Uniform Density (UD):** The size function \( S \) defines a size for each node \( n \). Treemap algorithms map nodes to hierarchical partitions of a subset of \( \mathbb{R}^2 \) such that the area \( A(n) \) of each partition is proportional to the size of the associated node \( n \). In other words: treemap algorithms ensure uniform density, where density is defined as size per area. Strict uniform density (SUD) requires that the density of all nodes is uniform, i.e.,

\[
\forall c, d : c \in V \land d \in V : \frac{S(c)}{A(c)} = \frac{S(d)}{A(d)}.
\]

while uniform density (UD) requires all nodes on the same depth to have uniform density, i.e.,

\[
\forall c, d : c \in V \land d \in V \land \text{depth}(c) = \text{depth}(d) : \frac{S(c)}{A(c)} = \frac{S(d)}{A(d)}.
\]

### 4.3. Generalized Treemap Algorithm

In this section we present a generalized treemap algorithm that can be parameterized into the base algorithms. We use pseudo code for presenting the algorithm.

A rectangular partition is defined as

```plaintext
rectangle = record
    xmin : double;
    xmax : double;
    ymin : double;
    ymax : double;
end;
```
The tree structure is stored as an object tree of `TreemapNodes`, which are defined as

```plaintext
TreemapNode = record
  parent      : TreemapNode;
  children    : list of TreemapNode;
  value  : double;
  rect   : rectangle;
end; .
```

All base treemap algorithms use a recursive structure. The exact implementation can differ, because it can be desirable to calculate all size values beforehand. Also, splitting up the algorithm has some performance advantages, because not all parts need to be calculated equally often. For simplicity and conciseness the algorithm is presented here as a single, depth-first recursion, though other implementations are viable as well, i.e.,

```plaintext
procedure Treemap(TreemapNode T; Bitmap B)
begin
  Render(B, T);
  Sort(T.children);
  Layout(T.children, [], T.rect);
  for each c Î T.children do Treemap(c, B);
{locally: SF Ù NO Ù IP Ù UD }
end; .
```

The treemap algorithm is initiated using a call to the `Treemap` method using the root node and a bitmap as parameters. Each node is drawn to the bitmap using the `Render` method. On termination of the algorithm, for each node of the tree, the `rect` property is defined, such that the `SF`, `NO`, `IP` and `UD` properties hold. `SF` ensures that all pixels of the bitmap are colored properly; there are which have undefined colors.

Sorting nodes (`Sort`) does not only accommodate the user, but is also used in the squarified treemap algorithm.

The layout algorithms of the base treemap algorithms differ. We now present a general layout algorithm that can be parameterized into the three base algorithms:

```plaintext
procedure Layout(list of TreemapNode children;
                  list of TreemapNode strip;
                  Rectangle R)
begin
  TreemapNode c := head(children);
  direction D := DirectionMethod(c, R);
  if AddNodeToStrip(strip, c, R, D) and c <> [] then
    Layout(tail(children), strip.add(c), R)
  else
    begin
      LayoutStrip(strip, R, D);
      if children <> [] then
        Layout(children, [], R \ Strip);
    end;
end; .
```
This procedure subdivides the rectangle by placing horizontal or vertical strips in the (remaining) rectangle. Nodes are added to strips using simple heuristics, which differ for the various algorithms. AddNodeToStrip contains these heuristics and indicates whether or not a node should be added to a strip, i.e.,

\[
\text{function AddNodeToStrip}(\text{list of TreemapNode strip}; \\
\qquad \quad \text{TreemapNode c}; \\
\qquad \quad \text{Rectangle R}; \\
\qquad \quad \text{Direction D}): \text{Boolean} \\
\begin{align*}
\begin{cases}
\text{Traditional: result := true;} \\
\text{Squarify : result := Worst(strip,R,D)\leq\text{Worst}(strip.add(c),R,D);} \\
\text{Strip : result := Avg (strip,R,D)\leq\text{Avg (strip.add(c),R,D);}} \\
\end{cases}
\end{align*}
\]

The heuristic that is used differs per base algorithm. The algorithm is greedy so the result is not always optimal; it is not guaranteed that optimal aspect ratios for a given ordered list of nodes are obtained. In [26], enhancements are proposed that add one-step look ahead information as input to the heuristics.

The traditional treemap algorithm always uses only a single strip, thus all nodes are added to this strip. The squarified treemap algorithm aims to optimize the worst aspect ratio in the strip; nodes are added to a strip as long as this improves the worst aspect ratio of the nodes in the strip. The worst function of the generalized treemap algorithm, i.e.,

\[
\text{function Worst}(\text{list of TreemapNode strip}; \\
\qquad \quad \text{Rectangle R}; \\
\qquad \quad \text{Direction D}): \text{double;}
\begin{align*}
\text{begin} \\
\text{LayoutStrip(strip, R, D);} \\
\text{result := 1;} \\
\text{for each c \in strip do} \\
\text{result := result min AspectRatio(c.rect);} \\
\text{end; },
\end{align*}
\]

does not make any assumptions on the strip in contrary to the worst function of the original squarified treemap which assumes that the nodes in a strip are descending in size. This enables more general use of the worst heuristic. The strip treemap algorithm uses a variation on the worst function, namely

\[
\text{function Avg}(\text{list of TreemapNode strip}; \\
\qquad \quad \text{Rectangle R}; \\
\qquad \quad \text{Direction D}): \text{double;}
\begin{align*}
\text{begin} \\
\text{LayoutStrip(strip, R, D);} \\
\text{result := 0;} \\
\text{for each c \in strip do} \\
\text{result := result + AspectRatio(c.rect);} \\
\text{result := result / strip.count;} \\
\text{end; },
\end{align*}
\]
which does not optimize for the worst, but for the average aspect ratio of the nodes in a strip.

If a node is added to a strip then the layout algorithm is called recursively and the next node will be evaluated. If a node is not added, then the strip is laid out and a new strip is started.

Laying out a *strip* uses the top or left part of rectangle $R$, such that the remaining area, if any, is a rectangle again. In case of a horizontal strip $\text{width}(\text{strip}) = \text{width}(R)$, in case of a vertical strip $\text{height}(\text{strip}) = \text{height}(R)$. The other side can be calculated using the desired density of the data, $\text{height}(\text{strip}) \cdot \text{width}(\text{strip}) = \frac{\text{size}(\text{strip})}{\text{Density}}$, which is for all nodes, and thus also strips, equal in case of the base treemap algorithms. This leads to

```plaintext
procedure LayOutStrip(list of TreemapNode strip;
                      Rectangle R;
                      Direction D);
begin
  if D=horizontal then
    R.ymax := R.ymin + size(strip)/(Density*width(R))
  else
    R.xmax := R.xmin + size(strip)/(Density*height(R));
  for each n ∈ strip do
    if D=horizontal then
      begin
        n.rect := R;
        n.rect.xmax := n.rect.xmin + size(n)/(Density*height(n));
        r.xmin := n.rect.xmax;
      end
    else
      begin
        n.rect := R;
        n.rect.ymax := n.rect.ymin + size(n)/(Density*width(n));
        r.ymin := n.rect.ymax;
      end;
  end;
end;
```

which is essentially the layout algorithm of the traditional treemap algorithm. So the traditional algorithm is used here as a building block of the other algorithms. Future work could look into layout algorithms that can be created using the proposed generalized treemap algorithm as a building block. Probably more optimal layouts can be generated with regard to aspect ratios, but on the other hand the structure will likely be less clear for users.

The direction of strips differs per algorithm. The traditional and strip treemap algorithms alternate between horizontal and vertical strips depending on the depth of the node in the tree. The squarified treemap algorithm always uses the shortest side of the remaining rectangle for a new strip, so the direction depends on the shape of the rectangle. This leads to
function DirectionMethod(TreemapNode T; rectangle R): Direction;
begin
  result := Horizontal;
  case TreemapAlgorithm of
    Traditional, Strip:
      if depth(T) mod 2 = 0 then result := Vertical;
    Squarified:
      if Width(R) > Height(R) then result := Vertical;
  end;
end;

which implements a choice that controls the second difference between the base algorithms. The last difference between the strip treemap algorithm and the squarified treemap algorithm is the sort order of the nodes. Nearly all implementations of the squarified algorithm sort the nodes on descending size. The strip treemap algorithm accepts all sort orders, thus creating order preserving layouts. This results in the following sort function

procedure Sort(list of TreemapNode children);
begin
  case TreemapAlgorithm of
    Traditional, Strip:
      SortOnUserRequest(children);
    Squarified:
      SortOnDescendingSize(children);
  end;
end;

Recapitulating the above, it can be said that the base algorithms differ on three points, namely
1. the sort order of the nodes (SO),
2. the direction method (DM),
3. the optimization criterion for adding nodes to a strip (OC).

At all three points a choice is made. The generalized treemap algorithm uses three parameters to control these choices and specialize the algorithm in either a traditional, a strip or a squarified treemap algorithm.

The base algorithms can be created using the following settings:

<table>
<thead>
<tr>
<th></th>
<th>SO</th>
<th>DM</th>
<th>OC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>Any sort order</td>
<td>Alternating</td>
<td>Add Always</td>
</tr>
<tr>
<td>Squarified</td>
<td>On Size Descending</td>
<td>Smallest</td>
<td>Worst</td>
</tr>
<tr>
<td>Strip</td>
<td>Any sort order</td>
<td>Alternating</td>
<td>Average</td>
</tr>
</tbody>
</table>

The generalized treemap algorithm allows all three choices to be made independently. Using other combinations of settings, results in variants on the base algorithms, like, for instance, using any sort order, the smallest direction method and optimizing for average aspect ratios.
5. Extending Treemaps

In Section 5.1, we first propose an extension to the generalized treemap algorithm as presented in Chapter 4. This extension allows for the creation of variations on the base algorithms that combine the strong points of the base algorithms without suffering from the weaknesses of these algorithms. Next, in Section 5.2, we propose additional extensions that enable the creation of treemaps which have the look and feel of well-known business graphics.

5.1. Level Based Layouts

The traditional treemap algorithm creates treemaps that have a structured and regular high-level layout, like for instance

![treemap with clear high-level layout.](image)

A drawback of the traditional treemap algorithm is that nodes can become very thin and elongated, and thus become difficult to compare and select. The horizontal white line in

![treemap, with selected node.](image)

indicates a node that is selected. Even worse, some nodes are even not visible anymore at all. The squarified treemap algorithm and the strip treemap algorithm solve this problem by creating squarer nodes, respectively,
While succeeding in creating fairly square nodes, both algorithms also lead to less structured treemaps.

So, recapitulating, the traditional treemap algorithm creates structured layouts, but also creates thin and thus confusing nodes. The squarified treemap algorithm and strip treemap algorithm create nodes that are more clearly visible, and easier to compare, but also result in a less structured layouts.

We propose mixed treemaps for combining the strong points of both algorithms, while avoiding the weak points. The combination of the traditional treemap algorithm and the squarified treemap algorithm allows for the creation of a treemap that has a structured layout, but also has clearly visible, squarified, nodes on the lowest level, i.e.,

The base treemap algorithms are instances of the generalized treemap algorithm that use particular combinations of parameters. Using different combinations of parameters within a single visualization leads to mixed treemaps.

In principle, the parameters could be set for each individual node. In practice this is unpractical because of the large number of nodes and the data dependency it would create. Therefore, we have chosen to set the parameters per level of the tree structure. For
each level the parameters can be set individually, so, like in the example, the lowest level can use a squarified treemap layout and the other levels a traditional treemap layout. The tree generator, which is also based on levels, creates tree structures all branches of which have equal depth. This makes the generated trees very suitable for visualization using a level based layout.

Furthermore, using levels has also been proven to be intuitive for users. Each level adds a layer to the generated tree structure and visualization that can be reasoned about separately.

Future work could research the possibilities of data driven parameters to obtain an optimal layout for each individual node.

5.2. Treemaps and Business Graphics

In this section we propose extensions to the generalized treemap algorithm which close the gap between treemaps and business graphics.

5.2.1. Size Method

The generalized treemap algorithm uses either a constant size or a size based on a data value for the leaf nodes of the tree structure $T=(V, E)$, i.e.,

$$S(n) = \begin{cases} 1 & \text{if } n_{children} = \emptyset, \\ \text{value} & \text{if } n_{children} = \emptyset. \end{cases}$$

The size of a higher level node $n$ is calculated by (recursively) taking the sum of the sizes of the child nodes of $n$, i.e.,

$$S(n) = \left( \sum_{c : c \in n_{children}} S(c) \right) \text{ if } n_{children} \neq \emptyset.$$

This approach ensures that UD holds; the density of all nodes is uniform. A consequence is that most of the time higher level nodes occupy different areas, like nodes A and B in treemap with high level nodes A and B.
However, often it is preferable that all nodes within a level use the same area, for instance when comparing ratios within the higher level nodes. Using the sum of the sizes of the children as size is an implicit choice of the generalized treemap algorithm. However, other choices are also viable. Therefore, we add other size functions on the higher levels, thus making the size function an explicit choice that can be set per level.

*Constant size* allocates exactly the same area for each node. The base algorithms only use constant size on the nodes level, we also allow constant size on higher levels,

\[
S(n) = 1 \quad \text{if} \quad n_{\text{children}} \neq \emptyset,
\]

when used on the top two levels of the same sample data, this leads to a treemap with *constant size* levels.

This matrix structure is useful for comparing two attributes of the data to each other; one attribute is mapped to the horizontal axis, the other to the vertical axis. It is clearly visible that constant size violates uniform density (UD) in this case, i.e., the yellow C nodes all represent the same quantity, but have different sizes.

*Average of children* calculates the average size of the children, i.e.,

\[
S(n) = \frac{\left( \sum_{c : c \in n_{\text{children}}} S(c) \right)}{|n_{\text{children}}|} \quad \text{if} \quad n_{\text{children}} \neq \emptyset.
\]

In general, area can effectively be used to represent average values. However, if the area is subdivided into smaller areas representing the child nodes, as is the case when using treemaps, then users intuitively expect this to represent the sum of the areas of the child nodes and not the intended average of the child nodes. This leads to confusing and even misleading views, therefore, this never should be used.

*Number of children* allocates space depending on the number of the children,

\[
S(n) = |n_{\text{children}}| \quad \text{if} \quad n_{\text{children}} \neq \emptyset,
\]
which is essentially equal to *sum of children* if the children all have equal sizes. If the children do not have equal sizes, then this suffers from the same problems as *average of children*; users interpret the area as the sum of the areas of the child nodes, not as the number of children.

*Sum of leaf nodes* is quite similar to *sum of children*,

\[
S(n) = \sum_{c : c \in V} c \land c_{\text{children}} = \emptyset \land n \in \pi_{B_{1, \ldots, B_{\text{depth}(n)}}} \left(\{c\} : S(c)\right)
\]

but instead of the child nodes all leaf nodes that are reachable from the node are summed. This is a useful heuristic for creating a level the size of which is independent from the level that is immediately below, i.e.,

resulting in more space for the rows of which the sum of the sizes of the leaf nodes is larger. Uniform density (UD) still does not hold, but the space is used more efficiently.

*Sum of category* works independently from the tree structure and sums the sizes of all leaf nodes that belong to the category of the current node, i.e.,

\[
S(n) = \sum_{c : c \in V} c \land c_{\text{children}} = \emptyset \land \pi_{B_{\text{depth}(n)}} \left(\{c\} : \pi_{B_{\text{depth}(n)}} \left(\{n\} : S(c)\right)\right)\text{ if } n_{\text{children}} \neq \emptyset
\]

leading to

resulting in more space for the rows of which the sum of the sizes of the leaf nodes is larger. Uniform density (UD) still does not hold, but the space is used more efficiently.
resulting in more space for columns the sum of the sizes of the leaf nodes of which is larger. Uniform density still does not hold. In section 5.2.8, we show how uniform density can be restored again.

The current heuristics for using space more efficiently are simple and, probably, can still be improved significantly. Future work can look into this.

5.2.2. Sort Method

Traditional treemaps and strip treemaps sort nodes to accommodate users in searching particular nodes. For small datasets this indeed helps users to find nodes quickly. For larger datasets it is not practical to search for nodes manually and it is best to provide search options to users [9]. This opens up the possibility of using the order of nodes for achieving visual effects. The squarified treemap algorithm uses this effectively to create a regular and ordered layout that does not distract users. We now present some sort methods.

Unsorted can be useful when data records, and thus nodes, contain an implicit order. However, if the number of nodes becomes larger, a traditional treemap algorithm becomes inconvenient because of the thin nodes, while a strip treemap leads to chaotic layouts if size and order do not correlate, for instance,

Sort by category sorts higher level nodes on the value of their category, both descending and ascending variants are available. On higher levels, the number of nodes is most often restricted, so this does not lead to the same problem, i.e., chaotic layouts, as sorting leaf nodes. This setting cannot be used in a squarified layout algorithm, because the squarified algorithm always uses sort by size descending.

Sort by size sorts nodes on their size; both descending and ascending variants are available. This results in layouts that are easier on the eyes. The squarified treemap and strip treemap algorithms using sort by size descending lead to
The ascending variant leads to an interesting variation on this. The squarified treemap and strip treemap algorithms using *sort by size ascending* lead to

In case of the squarified treemap algorithm, the big nodes are accumulated in the top left corner, not the small nodes. In case of the strip treemap algorithm, the big nodes are positioned at the top instead of the bottom.

This is another example where splitting the settings into levels is useful: nodes can be sorted on each level in a useful way. Examples can be found in Section 6.3.2.

### 5.2.3. Direction Method

The direction of a strip can either be horizontal or vertical. The traditional and strip treemap algorithm use alternating directions, which, in fact, also results in a per level setting: all odd levels have the opposite direction of the even levels.

Our generalization here is to set the direction for each level to horizontal, vertical or automatic. The added freedom of setting the directions manually leads to new variants.
Using the traditional treemap layout for the top two levels results in

\[ \text{traditional treemap, alternating direction,} \]

which makes only a precise comparison possible between either the top or the bottom quadrants. Setting the direction manually to vertical for both levels results in

\[ \text{treemap, only vertical direction;} \]

comparisons between all four groups are possible.

5.2.4. Desired Aspect Ratio

The base algorithms either result in very high aspect ratios or try to obtain aspect ratios close to one. While square nodes are in some sense optimal, they are not optimal in all cases. Text, for example, tends to have a larger width than height and thus nodes that are similarly shaped to this are more suitable [42]. The small nodes are too narrow to display numbers in
desired aspect ratio 1:1, while desired aspect ratio 1:1.9 suffers less from this because of a slight adjustment to the aspect ratio of the nodes.

The generation of other aspect ratios uses a distortion of the rectangle in which the nodes are laid out. Distorting a rectangle by virtually making the height bigger by a factor $\alpha$ and next creating square nodes leads to

\[
\begin{align*}
\frac{a \cdot h}{h} & = \alpha \cdot \frac{h}{h} \\
& = \alpha
\end{align*}
\]

Restoring the correct proportions of the rectangle,

\[
\begin{align*}
\frac{h}{a \cdot h} & = \frac{1}{\alpha} \\
& = \frac{1}{\alpha}
\end{align*}
\]

results in nodes with aspect ratio $\frac{1}{\alpha}$, if $\alpha > 1$.

This also opens up another way of looking at the traditional treemap layout: a traditional treemap layout is a squarified treemap layout that uses a desired aspect ratio of zero or infinity.
5.2.5. Vanishing Point

Plotting nodes of different sizes using the squarified treemap algorithm results in the well-known diagonal structure,

and contains a vanishing point in the top left corner. Different implementations choose different corners for the vanishing point.

Essentially, the squarified treemap algorithm adds nodes strip by strip to the layout. A strip is a row or column of nodes that uses all available space in respectively the horizontal or vertical direction. In the squarified treemap layout

first the blue strip is placed to the right, then in the remaining space the green strip to the bottom. This is repeated till all nodes are placed and thus the area is completely filled.

The location of the vanishing point depends on the edge the strips are added to. By altering the placement of the strips the vanishing point can be placed in each corner, i.e.,

Adding the strips alternating to opposite edges creates a layout,

in which the vanishing point is located, more or less, in the center.
Generalized, the attractor $a \in \mathbb{R}^2$ of the vanishing point can be chosen freely. A strip is added such that the center of the remaining rectangle is as close as possible to $a$, thus a strip is added to the edge that is most distant from $a$. The exact position of the vanishing point depends on the number and sizes of the nodes, but will most of the time be reasonably close to $a$.

Although very flexible, setting the vanishing point freely is overkill. Therefore, the user can choose between several presets. The location of the vanishing point is split into a horizontal and vertical component. For the horizontal component left, center and right are valid options, for the vertical component top, center and bottom, resulting in nine combinations. This provides enough freedom in most cases without overwhelming the user with settings. We already presented five combinations, the remaining combinations are

Reversing the sort order, thus sorting on ascending size, interchanges the positions of the small and big nodes, again leading to nine combinations, i.e.,

\begin{tabular}{ccc}
  \textit{Left} & \textit{Center} & \textit{Right} \\
  \textit{Top} & \textit{Top} & \textit{Top} \\
  \textit{Center} & \textit{Center} & \textit{Center} \\
  \textit{Bottom} & \textit{Bottom} & \textit{Bottom} \\
\end{tabular}

Bigger nodes are deemed more important, so positioning the bigger nodes in the center seems more logical. We have found no related work that does this.
5.2.6. Transformations

Pie charts are well known visualizations and are often already taught in primary school. This makes pie charts attractive visualizations; users do not have to learn how to use pie charts anymore. To create treemaps that resemble pie charts, we have added transformations in pixel space.

Transformations map points \( \mathbf{x} = (x, y) \) from a source rectangle \( R_s \) to points \( \mathbf{u} = (u, v) \) in destination rectangle \( R_d \). The coordinates of the source and destination rectangles are equal \( (R=R_s=R_d) \). Transformations are defined on the subset of \( \mathbb{R}^2 \) bounded by \(-1 \leq x, y \leq 1\), therefore, before the transformation the coordinates are normalized using

\[
\begin{align*}
\mathbf{n}(x, y) &= \left( \frac{2x}{\text{width}(R)} - 1, \frac{2y}{\text{height}(R)} - 1 \right)
\end{align*}
\]

and scaled back after the transformation using

\[
\begin{align*}
\mathbf{n}^{-1}(u, v) &= \left( \frac{u + 1}{2}, \frac{v + 1}{2} \right) \text{width}(R) \; \text{height}(R)
\end{align*}
\]

If differences in scaling between the \( \text{width}(R) \) and \( \text{height}(R) \) lead to undesirable distortions, then \( \min\{\text{width}(R), \text{height}(R)\} \) is used for scaling both coordinates.

Both forward \( \phi: \mathbf{x} \rightarrow \mathbf{u} \) and backward \( \phi^{-1}: \mathbf{u} \rightarrow \mathbf{x} \) transformations are necessary to enable interaction with the transformed visualization. The former is used for rendering the treemap, the latter is used for looking up which node is selected.

For each pixel \( \mathbf{x} \) of the destination bitmap a color is calculated by sampling the source bitmap in \( \mathbf{n}^{-1}(\mathbf{u}^{-1}(\mathbf{n}(\mathbf{x}))) \). Pixels that are not defined by the transformation are kept transparent so the parent layer remains visible.

Transformations in pixel space enable the creation of completely different looking visualizations from treemaps, like pie charts,

\begin{center}
\includegraphics[width=0.5\textwidth]{polar_transformation.png}
\end{center}

polar transformation of treemap,
but without creating new layout algorithms. Meanwhile the main advantage of treemaps is retained: both detail and overview remain visible. The base structure and semantics of the visualization are not changed, only the presentation differs.

Transformations in pixel space have several advantages. The main advantage is that the operation is completely independent of the other parts of the treemap algorithm. This means that everything else can remain rectangle based. Best of all, it is not necessary to create new layout algorithms, which is beneficial because layout algorithms are quite complicated, even for rectangular partitions.

For each level of a visualization a transformation can be defined. This enables creation of composite transformations by stacking the base transformations, resulting in intriguing visualizations, like

![transformed treemap](image)

that still have the benefits of regular treemaps: both detail and overview are present in one image, while the density remains uniform (UD). This example uses a pie transformation and two pyramid transformations, both are explained below.

Transformations should keep the density of the visualization uniform (UD) such that the area, and thus also the size, of nodes remains comparable. If UD holds before a transformation, then UD should also hold after the transformation. For a transformation \( \phi : x \to u \) this is the case if

\[
J_\phi(x, y) = \begin{vmatrix}
\frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} \\
\frac{\partial y}{\partial u} & \frac{\partial y}{\partial v}
\end{vmatrix} = c.
\]

For a composite transformation, UD holds if all base transformations maintain UD and the original, non-transformed, image has UD.
Transformations can map multiple source coordinates to one destination coordinate, like, for instance, the center point of a pie chart. The inverse transformation is not defined for such points. In practice this does not lead to problems, because the inverse transformation is only used for selecting nodes and a user does not associate those points with a meaningful selection. We now present several transformations.

*Identity* is the base transform; each pixel of the destination bitmap is mapped to the equivalent pixel of the source bitmap. This of course does not have any visual effect. It is important, however, because the original treemap algorithms without transformations are just special cases of the extended algorithm using the *identity* transformation.

\[
\phi(x, y) = (x, y) \\
\phi^{-1}(x, y) = \phi(x, y)
\]

*Flip horizontal* flips the pixels of the source bitmap horizontally. This option is also known as mirror horizontal. The effect can also always be mimicked using other options, i.e., changing the sort method from descending to ascending, but it is often more convenient to use a transformation for quickly inspecting flipped images.

\[
\phi(x, y) = (-x, y) \\
\phi^{-1}(x, y) = \phi(x, y)
\]

*Flip vertical* is the vertical counterpart of flip horizontal and flips the pixels of the source bitmap vertically.

\[
\phi(x, y) = (x, -y) \\
\phi^{-1}(x, y) = \phi(x, y)
\]
Pyramid squeezes one side of the rectangle, thereby creating a pyramid. Variants for all four sides are available, i.e., pyramid top, pyramid left, pyramid right, pyramid bottom. For pyramid top this leads to

\[
\phi(x, y) = \left(1 - \frac{y}{2}\right)x, y \right)
\]

\[
\phi^{-1}(x, y) = \left(\frac{2}{1 - y}\right)x, y \right)
\]

The above pyramid top transform does not keep the density of the visualization uniform. The top of the pyramid is squeezed more than the bottom of the pyramid; rectangles in the top become smaller than rectangles in the bottom. To compensate for this, we adjust the transformation, for pyramid top this leads to

\[
\phi(x, y) = \left(\frac{\sqrt{2(y+1)}}{2}\right)x, \sqrt{2(y+1)} - 1
\]

\[
\phi^{-1}(x, y) = \left(\frac{2}{y+1}\right)x, \frac{(y+1)^2}{2} - 1
\]

Pie Transform twists the input rectangle round the center point. This creates a pie chart.

\[
\phi(x, y) = (r \sin \alpha, r \cos \alpha),
\]

where

\[
\alpha = (x+1)\pi
\]

\[
r = \frac{y+1}{2}
\]

\[
\phi^{-1}(x, y) = \left(\frac{\alpha}{\pi} - 1, 2r^2 - 1\right),
\]

where

\[
\alpha = \arctan(x, y) + \frac{1}{2}\pi
\]

\[
r = \sqrt{x^2 + y^2}
\]

After this transform UD still holds. We also implemented a pie transformation that maps the y coordinate to the radius r of the pie.
Future research could explore the transformations space more exhaustively. Two interesting transformations that we have not discussed here are rotations and spirals. Rotations will not lead to new visualizations, but can be useful nevertheless. Spirals can be used for creating a continuous axis that is longer than the screen width or height.

5.2.7. Margins

The hierarchical structure of

![treemap without margins](image1)

is accentuated in

![nested treemap](image2)

by using margins (also known as borders), resulting in nested treemaps [1]. Margins use some of the area of nodes, thus leaving less area available for the child nodes. Related work uses fixed margins that have the same width on all four sides [1, 42], often based on the depth of the node in the tree structure [21, 23, 26].

We generalize this, such that, for each level, the width or height of the margins can be individually set for the left, top, right and bottom margins, both in absolute pixel values and percentages of the rectangle. For each node $n$ a rectangle $n\.childrenrect$ is created inside $n\.rect$ that is used for laying out the children of $n$, i.e.,

![overview of margins](image3)
Using margins in general violates UD. In related work, small, absolute margins are used and the change in density this causes is neglected [1, 21, 23, 26, 42]. However, even small, absolute margins can result in empty or degenerative children rectangles, thus leading to disappearing leaf nodes. Relative margins that are fractions of the nodes do not suffer from this problem.

5.2.8. Adjust for Uniform Density

Uniform density (UD) is an important property of the treemap algorithm because it enables quantitative comparisons of nodes. Some of the newly proposed settings violate UD, most notably margins and all size methods other than sum of children.

Treemaps that do not conform to UD are nearly always misleading. In our experience, users intuitively interpret area as the most important visual property for associating nodes with quantitative values [45]. Even if users are explicitly told to ignore the area and focus, for example, on the width or height of nodes, this still results in confusion. Therefore, UD is regarded as an important property of treemaps that should hold for all levels the sizes of which represent quantities.

To adjust for UD, per level \( L \) of the tree structure \( T=(V, E) \), the user can choose whether \( UD_L \) should hold, where

\[
UD_L : \left( \forall p, r : p \in V \land r \in V \land \text{depth}(p) = \text{depth}(r) = L : d_p = d_r \right),
\]

where

\[
d_n = \frac{\sum_{c : \text{parent}(c) = n : S(c)}}{a(n_{childrenrect})}
\]

So, \( UD_L \) holds if, for each two nodes \( p \) and \( q \) of level \( L \), the densities of the child nodes of \( p \) and \( q \) are equal. At this stage, the areas of nodes of level \( M=L+1 \) are not known, because the layout of level \( M \) is not yet calculated. However, the sizes of the nodes of level \( M \) are available. Therefore, the density \( d_n \) of a node \( n \) is calculated by dividing the sum of the sizes of the child nodes by the area of the children rectangle \( n_{childrenrect} \).

The size \( S(c) \) of a node \( c \) cannot be adjusted. To change \( d_n \), only the area \( a(n_{childrenrect}) \) can be changed. Enlarging \( n_{childrenrect} \) is not an option as this could violate the inside parent (IP) and/or non-overlapping (NO) properties. So the only remaining option is to make \( n_{childrenrect} \) smaller and thus \( d_n \) larger.

To make \( n_{childrenrect} \) smaller, additional margins are added to level \( L \). The top margin, left margin, right margin and bottom margin can all independently be set to auto-size. A margin that is set to auto-size is enlarged if \( n_{childrenrect} \) must be decreased. For instance, auto-size on the top margin leads to
To calculate the sizes of the auto-size margins, first, calculate the maximum density $d_{\text{max}}$ of level $L$, i.e.,

$$d_{\text{max}} = \max \{ d_p \mid p \in V \land \text{depth}(p) = L \}.$$

Next, for each node $n$ of level $L$, adjust the auto-size margins such that the density $d_n$ of the child nodes becomes equal to the maximum density $d_{\text{max}}$.

The next examples show a fixed $M_{\text{top}}$ margin in blue, without and with an auto-size margin in red, respectively.

All leaf nodes have equal size; it is clearly visible that UD does not hold when only a fixed margin is used. The auto-size margin corrects for this, resulting in a bar chart.
5.2.9. Draw Line to Parent

Treemaps are useful for large trees, but for expressing the structure of small trees, node-link diagrams are often more suitable. Users are more accustomed to node-link diagrams than to the equivalent treemap.

To bridge the gap between treemaps and node-link diagrams we add to treemaps the option to draw lines on top of the treemap; each node can optionally draw a line to its parent.

For each level, a user can set whether lines are drawn or not and what connection points are used. If lines are drawn for a level, then for each node of the level, a line is drawn from the connection point of the node to the connection point of the parent node.

The $x$ and $y$ positions of the connection points can be set independently. The $x$ position can be set to left, center and right. The $y$ position can be set to top, center and bottom. This way, nine connection points can be set, i.e.,

![Connection points for lines.](image)

Extending treemaps with lines enables node-link diagrams to be created without creating new layout algorithms. Furthermore combinations between treemaps and node-link diagrams become possible, like
This example uses top margins to obtain an icicle plot layout which is further enhanced using lines. Combinations like this can combine the strong points of node link diagrams and treemaps: representing high level structure using lines and using squarified leaf nodes to prevent clutter.

5.2.10. Matrix Layouts

The squarified treemap algorithm generates a space filling layout of 13 equally big nodes,

in which the density of all nodes is uniform, the shape of the nodes, however, differs. This is undesirable if impartial representation of entities is needed, for instance, in case of educational data, all students are equal and should be presented exactly the same to avoid bias.

This requires all nodes to have the same dimensions. In case of 13 nodes, it is not possible to create a matrix layout while maintaining a space filling layout. Therefore, we allow non-space filling layouts, created by adding empty nodes. Empty nodes are not associated with any data record, they are place holders that are used to obtain better layouts. Adding 3 empty nodes and using the squarified and strip treemap layouts, leads to, respectively
We call this *matrix* layouts.

To create a matrix layout of $N$ nodes in a rectangle $R$, we have to choose the number of rows $r$ and the number of columns $c$, such that $r \cdot c \geq N$, resulting in $N_{\text{empty}} = r \cdot c - N$ empty nodes. To avoid wasting space, we limit the number of empty nodes to $N_{\text{empty}} < c$, so there are no rows or columns all cells of which are empty.

All nodes have the same width and height, for each node $n$, $\text{width}(n) = \frac{\text{width}(R)}{c}$ and $\text{height}(n) = \frac{\text{height}(R)}{r}$.

Thus all nodes have equal aspect ratio $\alpha = \alpha_R \cdot \frac{r}{c}$, where $\alpha_R = \frac{\text{width}(R)}{\text{height}(R)}$.

We now choose $r$ and $c$ such that the aspect ratio $\alpha$ is as close as possible to the, user defined, desired aspect ratio $\alpha_d$, i.e., choose $r$ and $c$ such that $|\alpha - \alpha_d|$ is minimal.

The matrix layout has at least 1 row and at most $N$ rows, i.e., $1 \leq r \leq N$. For a given number of rows $r$, the number of columns $c = \left\lceil \frac{N}{r} \right\rceil$, which satisfies $r \cdot c \geq N$ and $N_{\text{empty}} < c$.

Thus, to create a matrix layout of $N$ nodes in a rectangle $R$, calculate $\left| \alpha_R \cdot \frac{r}{\left\lceil \frac{N}{r} \right\rceil} - \alpha_d \right|$ for each row $r$, $1 \leq r \leq N$. Finally, choose $r$ such that the calculated value is minimal.
5.2.11. Empty Nodes

Matrix layouts can also be created by using two levels using traditional layouts, one using horizontal divisions and one using vertical divisions, representing, respectively, the rows and the columns of the matrix.

However, if columns contain a different number of rows, or rows contain a different number of columns, then no matrix structure will emerge. Adding empty nodes can counter this by making sure each column has an equal number of rows or each row has an equal number of columns. First, using a vertical traditional layout and next a horizontal traditional layout, without and with empty nodes, leads to

![Treemap without empty nodes](image1)

![Treemap with empty nodes](image2)

A tree $T=(V,E)$, created from the relation $D$ using attributes $a_1, \ldots, a_n$, leads to matrix layouts for level $i-1$ and level $i$ if

$$C_i : \pi_{B_{i-1},\ldots,B_n}(D) \times \pi_{B_i}(D) = \pi_{B_{i-1},\ldots,B_n}(D)$$

holds, where $\times$ is the Cartesian product. The Cartesian product generates a set of tuples for each combination of the two input tuples, like, for instance

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>R x S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>City</td>
<td>Name</td>
</tr>
<tr>
<td>Dennis</td>
<td>Munstergeleen</td>
<td>Present</td>
</tr>
<tr>
<td>Jake</td>
<td>Eindhoven</td>
<td>Absent</td>
</tr>
</tbody>
</table>

To generate a tree $T_{C_i} = (V_{C_i}, E_{C_i})$, that results in a matrix layout for levels $L_{i-1}$ and $L_i$, we add empty nodes till $C_i$ holds, i.e.,

$$V_{C_i} = V \cup \pi_{B_{i-1},\ldots,B_n}(D) \times \pi_{B_i}(D)$$

$$E_{C_i} = E \cup \{ (c, p) \mid c \in \pi_{B_{i-1},\ldots,B_n}(D) \times \pi_{B_i}(D) \land p \in \pi_{B_{i-1},\ldots,B_n}(\{c\}) \}.$$
6. Results

The generalized and extended treemap algorithm presented in the previous chapters contains many parameters, such as layout algorithms, size functions and transformations, mostly on a per level basis. Nearly all parameters are independent of each other, resulting in many possible combinations. Each combination of parameters generates a different visualization, some useful, some not, often dependent on the questions being asked and the data being visualized. This chapter first presents the criteria we have determined to be important when assessing visualizations in Section 6.1. In Section 6.2 the structure of our visualizations is analyzed. In Section 6.3 some real world examples are presented. Finally, in Section 6.4 we present some rules of thumb.

6.1. Requirements

It is not easy to characterize a good visualization, because it not only depends on the intended use of the visualization, but also depends on the data used. Visualizations are often used to:
- explore unknown relationships or outliers in data,
- examine expected relationships in data,
- interact with the underlying data records,
- summarize conclusions of the data and
- present conclusions to others.

Each particular use poses different requirements on a visualization. Exploring data, for example, requires an overview of the data because it is not certain where interesting properties are located. Interaction with the data records, on the other hand, requires a one-on-one mapping between data records and visual entities in the visualization.

In practice, however, the same visualization is often used for more than one purpose. First, a dataset is explored, next, some uncovered relationships are summarized, which are finally presented in a meeting. So, ideally a visualization should facilitate all different use patterns during its life cycle.

Visualizations are used for visualizing many different kinds of input data. Furthermore, different tree structures can be generated from the input data, even using the same input data. The properties of these tree structures determine which visualizations are suitable. The number of child nodes, for instance, is an important factor in deciding which layout algorithms to use. It is not possible to create a single visualization that is optimal for all possible tree structures.

Nevertheless we found that for all visualizations similar requirements apply. The most important are:
- Uniform density is required for quantitative analysis of data. Other properties can be used to encode quantitative information, but these are intuitively overruled by
size. Uniform density on the records level makes comparison of values possible; comparison of aggregates also requires uniform density on the other levels.

- A one-on-one mapping between records and visual entities is required for looking up values and interaction with the data. Although an aggregate can be represented using a single visual entity, this leads to ambiguous semantics when interacting with the aggregate.

- An overview helps users to put things in perspective; all data is presented in the same view, no data is concealed. This makes a visualization more transparent and enables explorative data analysis.

- The structure of the data should be clearly visible. Categorizing similar entities results in groups the properties of which become visible. Users can focus on particular relationships by choosing different groupings.

- Familiarity leads to lower learning curves and, in general, a more positive user response. Therefore, resemblance to well-known business graphics is considered to be an important advantage.

- Aesthetically pleasing visualizations are more rewarding for users. This leads to more motivated users that are willing to invest more time in learning the visualization, thus resulting in better use of the visualization in general.

6.2. Analysis

The parameters of the presented generalized and extended treemap algorithm are set per level, i.e., for all nodes that have the same depth. Depending on the type of level different settings are most often used. We distinguish three different kinds of levels in a visualization:

1. **Records levels** are the deepest level of a visualization. All nodes of these levels uniquely represent records. Values of attributes of the records are mapped to the color used to present the nodes.

2. **Structural levels** group individual graphics into new graphics that still contain full detail of the individual graphics, but also use structure for displaying overviews and aggregates. Structural levels can be used, for example, to structure the nodes of the records level into bar charts or pie charts.

3. **Layout levels** define the global layout of sets of graphics that are perceived by users as separate entities. The position of these graphics has no real meaning to the user. Layout levels provide overviews of the data and enable users to select interesting subsets in a natural way by navigating through the data.
6.2.1. Records Level

All visualizations have a records level. Each node of this level represents one data record uniquely, such that a one-on-one mapping between nodes and data records is maintained. The one-on-one mapping makes it possible to interact with the records in the information system, which is a unique advantage of treemaps that standard business graphics lack.

Square nodes are most suitable for interaction, because they are most easily selected. Therefore, on the records level, the squarified layout algorithm is nearly always used. Depending on the location of the vanishing point and the order of the nodes, there are 18 variations as shown in Section 5.2. The most often used variations are

- squarified treemap, “big nodes in center”
- squarified treemap, “small nodes in corner”.

The former has the advantage of showing the more important nodes in the center, the latter is more suitable for showing a correlation between size and color. However, it seems that user preference plays an equally important role.

Sometimes, particularly when transformations are used, the strip layout algorithm is preferable because this leads to quieter layouts. This is clearly visible when the polar transformation of a squarified and a strip layout are compared, respectively

- squarified layout, polar transformation,
- strip layout, polar transformation.
Different aspect ratios can be used to further enhance the layout. Generating a pie of a strip layout using a desired aspect ratio of 1 and 0.16 leads to

6.2.2. Structural Levels

Structural levels group individual graphics into new graphics that still contain all detail, but also use structure for displaying overviews and aggregates. Among these newly created graphics are familiar business graphics like pie charts and bar charts.

To emphasize structure, we recommend to use traditional treemap layouts for structural levels, in contrast to the records level where squarified treemap layouts are preferable. This again is a clear confirmation that multiple layout algorithms in one visualization are beneficial; the layout algorithms have different uses, combining them is synergetic.

Each structural level uses a traditional treemap layout algorithm to divide the available space in either horizontal (H) or vertical slices (V), respectively.

If more structural levels are used, for each level the division direction can be set separately to either horizontal or vertical. If two successive structural levels use the same division direction then they are called parallel, otherwise they are called orthogonal.
Starting with the vertical layout $V$, we can add another structural level either parallel ($VP$) or orthogonal ($VO$), resulting in

To both $VP$ and $VO$ a third structural level can again be added parallel or orthogonal. For $VP$ this results in $VPP$ and $VPO$, i.e.,

For $VO$ this results in $VOP$ and $VOO$, respectively
Note that the order in which parallel and orthogonal levels are added is important. First, adding a parallel level and then an orthogonal level is not the same as first adding an orthogonal level and then a parallel level, in the above example VOP, for instance, is not the same as VPO.

Adding only orthogonal levels results in traditional treemap layouts, i.e. VO and VOO. Both VO and VPO are essentially 100% stacked columns as found in, for example, Microsoft Excel and statistical graphics.

Using only parallel levels results in visualizations that are similar to icicle plots, i.e. V, VP and VPP. This can be made clearer by adding borders to the top of each structural level and the root level, i.e.,

```
V.
V
VV
VP
VVV
VPP.
```

Parallel levels lead to layouts as used in icicle plots, tree diagrams and list views. In these layouts hierarchical structures are mapped to a single dimension (1D). Comparing sizes simply becomes comparing the widths of columns or heights of rows. Order can naturally be mapped to a linear horizontal or vertical axis. However, there is a limit on the size of the structures that can be displayed in this way.

Adding orthogonal levels maps the hierarchical structure to two dimensions (2D) and enables displaying larger hierarchical structures. However, structural information is perceived less strongly when using orthogonal representations. An additional advantage orthogonal layouts offer is better quantitative comparisons of more than one attribute, for instance comparison of the relative importance of the blue component in VP and VO is much clearer in VO. A disadvantage of orthogonal layouts is that comparing sizes becomes dependent on both the width and the height of a node, leading to less accurate readings [37].

Structural levels can use constant size to make the structure more visible. Originally, treemap algorithms use the sum of the sizes of the child nodes for the size of inner nodes, which can causes small categories to be displayed very small or disappear completely. Sometimes this is desirable, because small categories are not deemed important, sometimes all categories are considered equally important, so they should be displayed equally large.
Changing the size function of the top level to constant size results in

Because there are two categories on the top level, the display space is split exactly in half. Changing the size function of the second level also to constant size results in

Finally, changing the size function for all structural levels to constant size leads to

The use of constant size results in columns and rows of equal width and height. In $VPO$, $VOP$ and $VOO$ matrix like structures emerge. This is the result of two levels that are orthogonal to each other and have constant size. Matrices can be desirable if the number of categories is low and the resulting matrix structure is not too sparse.

If not all subcategories are present for each category then the matrix structure collapses visually, adding empty nodes resolves this. This is clearly visible in the example below, respectively without and with empty nodes,
Constant size in general conflicts with uniform density, i.e., the sizes of the nodes are not comparable anymore. As explained in Section 5.2, we correct for this by adding borders and thus not using all available space in a node for child nodes.

Adding a border to the top of $V$, such that uniform density is guaranteed, results in a bar chart, for $VO$ this results in a stacked bar chart, respectively

Stacked bar charts are useful for comparing totals, comparing the subcategories is more difficult because they are not aligned.

Starting with $VP$ using constant size on the top two levels and adding borders to the top, right and all four sides of the second level, results in
The first image is available in Microsoft Excel as a *clustered column*, this is also known as a parallel bar chart. In contrary to stacked bar charts, parallel bar charts are less suitable for comparing totals, but more suitable for comparing the sizes of subcategories. Parallel bar charts become very thin and elongated if the number of bars is big, just like the traditional treemaps and icicle plots, because all categories are mapped to a single axis. The second and third images are only presented for completeness, they are not particularly useful.

Starting with *VO* using constant size on the top two levels and adding borders to the top, right and all four sides of the second level, results in

![Image](image1.png)

VO top border, VO right border, and VO all borders.

Depending on what data is visualized, these variations can be useful. Using *VO* with a top border, subcategories can be compared *between* categories; using a right border they can be compared *within* categories, using all four borders makes both comparisons *between* and *within* categories possible. In the last case, however, quantitative resolution is lost because size depends on the product of width and height. In all cases the total of categories is more difficult to compare than in case of a stacked bar chart; all variations have their advantages and disadvantages.

Structural levels can use *transformations* to obtain a different look, for instance pie charts. Transformations are completely orthogonal to the other settings. A polar transformation, for example, turns

![Image](image2.png)

traditional layout (1 level) into pie chart,
Polar layouts, i.e., pie charts, have several characteristics:
- pie charts are useful for conveying particular ratios, i.e., whether a ratio is slightly lower or higher than 50% or 25% is clearly visible,
- polar layouts are more space efficient when displaying leaf nodes of trees in 1 dimension, i.e., the circumference of a polar layout is $\pi$ times larger than the height or width,
- pie charts are preferred by some users, either because they are familiar with them or because they find them aesthetically pleasing,
- pie charts are perceived as a single graphic, i.e., two adjacent pie charts are clearly recognizable as two graphics, see also Section 6.2.3. This in contrast to two adjacent squarified treemaps, which blend together. The advantage hereof is that many pie charts can be displayed besides each other without causing confusion. A disadvantage is that comparison of quantities between two pie charts is difficult,
- errors are higher when reading values from pie charts, than when reading values from bar charts [37].

Structural levels can use lines to emphasize tree structures. Just like transformations, lines are independent of the other settings. Adding lines results in images like
Lines are most useful when categories are mapped to a single axis. Using lines together with a squarified layout leads to overlapping lines. This can lead to clutter, but can also be beneficial because the other axis can also be used for displaying structure.

Because nearly all settings are independent of each other, they can be combined in many ways, for example, using

This way the strong points of each setting can be used, resulting in visualizations that are more suited to the data being used and the questions being asked.
6.2.3. Layout Levels

Layout levels, in contrast to structural levels, do not position categories on horizontal or vertical axes to aid in quantitative comparisons. Layout levels pack multiple graphics to a single visualization. Multiple pie charts can, for example, be displayed using

![Pie charts in a squarified layout level](image1)
or

![Pie charts in a matrix layout level](image2)

When using a layout level, hundreds of pie charts can be displayed in a single screen. This provides an overview which can be used to search for outliers and navigate to pie charts of interest. Quantitative comparison of subcategories displayed using a layout level is often difficult because (sub)categories are often not aligned. In case of a squarified layout level this is very clear, in case of a matrix layout level, some nodes are aligned, but not all nodes are directly comparable to each other.

Visualizations can use multiple layout levels. The following example uses two layout levels, both squarified layouts, i.e.,

![Pie charts in two squarified layout levels](image3)

In this example, one structural level is used for creating the pie charts. The records level uses a strip layout. Quantitative comparison of areas is not possible between pie charts; uniform density is only guaranteed within a single pie chart. Future work should look into this.
6.3. Real World Examples
This section shows some real world examples of visualizations we have created and used in presentations in the past.

6.3.1. Cases of a Notary’s Office
Each cushion in this example is a case of a notary’s office. A green cushion is a profitable case; a red cushion represents a loss-making case. The larger the cushion, the larger the profit or loss.

<table>
<thead>
<tr>
<th>level</th>
<th>layout</th>
<th>squarified</th>
<th>product group</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>level 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>level 3</td>
<td>records</td>
<td>squarified</td>
<td>product</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>case.</td>
</tr>
</tbody>
</table>

This is one of the first visualizations we created and only squarified layouts are used. It is clearly visible that the “Familiepraktijk” contains some subcategories that make a loss. However, because no structural levels are used, it is impossible to read whether the “Ondernemingspraktijk” or the “Onroerend goed praktijk” has a bigger proportion of profitable cases. Using extended treemaps we can create new visualizations of this dataset and choose what structure should be emphasized. For instance, to show the proportion of profitable cases for the product groups “Ondernemingspraktijk”, “Familiepraktijk” and “Onroerend goed praktijk”, we can create this visualization
cases of a notary’s office

<table>
<thead>
<tr>
<th>level 1</th>
<th>structural</th>
<th>traditional</th>
<th>product group</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 2</td>
<td>structural</td>
<td>traditional</td>
<td>profit/loss</td>
</tr>
<tr>
<td>level 3</td>
<td>layout</td>
<td>squarified</td>
<td>product</td>
</tr>
<tr>
<td>level 4</td>
<td>records</td>
<td>squarified</td>
<td>case</td>
</tr>
</tbody>
</table>

For showing the proportion for all individual products, we can create this visualization

cases of a notary’s office

<table>
<thead>
<tr>
<th>level 1</th>
<th>structural</th>
<th>traditional</th>
<th>left margin</th>
<th>product group</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 2</td>
<td>structural</td>
<td>traditional</td>
<td>left margin</td>
<td>product</td>
</tr>
<tr>
<td>level 3</td>
<td>structural</td>
<td>traditional</td>
<td>left margin</td>
<td>profit/loss</td>
</tr>
<tr>
<td>level 4</td>
<td>records</td>
<td>squarified</td>
<td></td>
<td>case</td>
</tr>
</tbody>
</table>

The latter two examples make quantitative comparison of products and product groups easier, but the sense of detail is less prevalent. None of the examples are superfluous; all tell a different story.
6.3.2. Grades of a High-School

Each cushion in this section represents a grade of a high-school student. Each cushion has the same size, i.e., each grade is deemed equally important. The color of a cushion indicates the grade: red is an unsatisfactory grade (5.5 or below), yellow is a high grade (9 or higher), the other colors represent grades in between.

The first example shows the distribution of grades per subject, i.e.,

<table>
<thead>
<tr>
<th>level 1</th>
<th>structural</th>
<th>traditional</th>
<th>subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 2</td>
<td>structural</td>
<td>traditional</td>
<td>grade bin</td>
</tr>
<tr>
<td>level 3</td>
<td>records</td>
<td>squarified</td>
<td>grade</td>
</tr>
</tbody>
</table>

To compare all subjects, it is best to create a structural level using a traditional layout. This is feasible because the number of subjects is limited, namely 20. If the number of subjects would have been significantly higher, using a traditional layout would not have been feasible, because the columns would become too narrow. A vertical layout is used because this is better for comparing quantities [37].

To make the distribution visible an orthogonal structural level is used. The orthogonal layout makes comparing the grades between subjects possible. Again the number of categories (five) permits using a traditional layout for this level.
The second example compares the distribution per subject, per gender of the teacher and gender of the student, see also Section 5.2.3, i.e.,

<table>
<thead>
<tr>
<th>level</th>
<th>layout</th>
<th>squarified</th>
<th>subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 1</td>
<td></td>
<td></td>
<td>subject</td>
</tr>
<tr>
<td>level 2</td>
<td>structural</td>
<td>traditional</td>
<td>gender teacher</td>
</tr>
<tr>
<td>level 3</td>
<td>structural</td>
<td>traditional</td>
<td>gender student</td>
</tr>
<tr>
<td>level 4</td>
<td>structural</td>
<td>traditional</td>
<td>grade bin</td>
</tr>
<tr>
<td>level 5</td>
<td>records</td>
<td>squarified</td>
<td>grade.</td>
</tr>
</tbody>
</table>

Within each subject, from left to right,
- first, the grades given by male teachers to male students are displayed,
- second, male teachers to female students,
- third, female teachers to male students and
- fourth, female teachers to female students.

Some subjects only have teachers from one gender and thus only show two nodes instead of four. To compare all four categories with each other, two parallel structural levels are needed. If orthogonal levels had been used, then not all six combinations could have been compared directly, see also Section 5.2.3.
It would have been nice if the subjects could also have been added using a parallel structural level. However, the number of parallel columns would have become too large, namely 58, thus resulting in very thin columns. Using an orthogonal structural level to create a matrix layout does not add information in this case, therefore, a squarified layout level is used to show all subjects.

To compare the grades between subjects and to compare the global relation between gender and grades, separate views should be used, like the example above.

The third example compares the grades per class, both year and school type, per subject and per teacher. A pie chart contains all grades a teacher gave to the students in a school year and school type, i.e.,

<table>
<thead>
<tr>
<th>level 1</th>
<th>structural</th>
<th>traditional</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 2</td>
<td>structural</td>
<td>traditional</td>
<td>school type</td>
</tr>
<tr>
<td>level 3</td>
<td>layout</td>
<td>matrix</td>
<td>subject</td>
</tr>
<tr>
<td>level 4</td>
<td>layout</td>
<td>matrix</td>
<td>teacher</td>
</tr>
<tr>
<td>level 5</td>
<td>structural</td>
<td>traditional</td>
<td>polar transformation</td>
</tr>
<tr>
<td>level 6</td>
<td>records</td>
<td>strip</td>
<td>grade.</td>
</tr>
</tbody>
</table>

grades per year and school type, per subject and per teacher
The year and school type of a class both have a limited number of different values, therefore, they can be visualized using orthogonal structural levels. A matrix like structure is visible. It is chosen not to use empty nodes, because of the number of nodes and a lack of space. The structure remains very clear, however, thus allowing the user the zoom quickly to interesting classes.

Within each year and school-type all subjects are shown. The number of subjects multiplied by either the number of school types or years is too big to use a structural level. All subjects are deemed equally important, so a matrix layout is chosen over a squarified layout. This has the added advantage of leading to a quieter layout. For the same reasons, within each subject, a matrix layout level is chosen for displaying all teachers. The pie charts are polar transformations of a structural level which groups all marks just like in the previous examples.

The final example of the high-school dataset shows all students of class H5 and their history of classes together with the distribution of their grades. The selected student started in class BK and moved up via A2, A3 and H4 to H5, i.e.,

<table>
<thead>
<tr>
<th>level 1</th>
<th>structural</th>
<th>none</th>
<th>polar transformation</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 2</td>
<td>structural</td>
<td>traditional</td>
<td>top margin</td>
<td>H5</td>
</tr>
<tr>
<td>levels 3-8</td>
<td>structural</td>
<td>traditional</td>
<td>top margin, lines</td>
<td>history</td>
</tr>
<tr>
<td>level 9</td>
<td>structural</td>
<td>traditional</td>
<td></td>
<td>grade bins</td>
</tr>
<tr>
<td>level 10</td>
<td>records</td>
<td>squarified</td>
<td></td>
<td>grade</td>
</tr>
</tbody>
</table>
In this example lines are used on top of the treemap to enhance the paths the students move up through H5. A radial layout was chosen because a Cartesian layout was too cluttered.

6.3.3. XML Structured Text

This section uses an XML structured text as example. The text is partitioned into chapters, sections, headers, sub-sections and individual lines. The color of a cushion represents the type of a text block; yellow, for instance, indicates headers. The size of a cushion represents the size of a text block. This results in

<table>
<thead>
<tr>
<th>level 1</th>
<th>structural</th>
<th>traditional</th>
<th>top margin</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 2</td>
<td>structural</td>
<td>traditional</td>
<td>top, left, right margin</td>
<td>Section</td>
</tr>
<tr>
<td>levels 3-5</td>
<td>structural</td>
<td>traditional</td>
<td>Subsection</td>
<td></td>
</tr>
<tr>
<td>level 6</td>
<td>records</td>
<td>traditional</td>
<td>text line.</td>
<td></td>
</tr>
</tbody>
</table>

In this case only structural levels are used. The first level is horizontal to facilitate the placement of labels. Constant size is used to create a rigid chapter structure.
The second level is orthogonal so the second dimension can be used. Note that more traditional methods for displaying structured text, like tree views, only use parallel levels, resulting in vertical scrollbars because there are too many nodes to fit on a single screen.

The third level is parallel because an orthogonal level would result in too many thin horizontal strips. This typically is a case where the nature of the input data prescribes a particular visualization. To keep a rigid structure, the structure is reinforced using borders.

Finally, the fourth level uses an orthogonal layout. This not only leads to a clearly visible structure without using borders, it also results in a horizontal orientation for the final level, which is beneficial because the leaf nodes contain text. This text becomes visible if a user navigates, i.e., zooms, to a particular sub-section. Navigation is easy because of the highly structured representation of the text.

### 6.4. Rules of Thumb

In this section we present a list of rules of thumb we have found useful when creating visualizations.

- **Use introduction views.**
  When creating a view set, first, create a view that uses a squarified layout for the records level and has no other levels. Do not use color or size in this view. Explain what a cushion represents. Next choose the size and color attributes for the view set and create two views that explain what the color and size of cushions represent. Repeat this if the color or size changes.

- **Use color prudently.**
  Colors, perhaps more than anything else, determine the appearance of a visualization. Attractive and eye-catching colors should be used. Saturated colors can be used to draw attention to interesting nodes. Using too many saturated colors should be avoided. In general about eight colors can be used in a single visualization; using more colors does not lead to better visualizations. Make sure colorblind people can distinguish and preferably also name the colors. Use a structural level to partition colors of the records level into categories. This leads to quieter layouts and better comparisons of aggregated areas of colors.

- **Use a squarified layout on the records level.**
  Squarified layouts results in the best aspect ratios, which is important on the records level. When fine-tuning visualizations, changing the sort order or aspect ratio can be beneficial, as can be the use of a strip treemap layout. Traditional treemap layouts are normally not used on the records level. Matrix layouts should not be used on the records level, because they can lead to distortions of size.
- **Use as many structural levels as possible.**
  More structure is in general preferable, because it gives a clearer and quieter image. A structural level can map about 20 categories to one dimension. Using orthogonal structural levels, about 20 categories can be added to each dimension. This makes visualizations more scalable, but still very structured. Displaying too many categories leads to thin nodes, which are difficult to handle. In this case, use squarified or matrix layout levels to generate squarer nodes.

- **Use matrix layouts when squarified layouts result in clutter.**
  Squarified layouts are suitable for displaying many nodes that have different sizes. This can lead to clutter. Use matrix layouts if the different sizes are not important, for instance, when comparing ratios.

- **Only use radial layouts when appropriate**
  Use radial layouts to display about three times more nodes using a single dimension. Do not use radial layouts if quantitative comparison of nodes is important.

- **Mimic the natural order of categories.**
  If categories have a natural order, then this order should also reflect in the visualization. This helps users in navigating the visualization and prevents confusion.

- **Quantities should be comparable.**
  Uniform density should always hold. Preferably, a single axis is used to represent quantity as this enables comparison on height or width. Using two axes does not lead to confusion, but the accuracy of reading values is lower.

- **Enhance structure.**
  First create views with default shading and without borders, lines and empty nodes. Add empty nodes if this enhances the structure without wasting too much space. Next, adjust the shading such that the overall visualization is not too dark and relevant hierarchical structure remains visible. If the structure is already clear due to the layout used, then the shading can be turned down. Add borders if shading is not sufficient. If the number of categories is small (about ten or less) lines can be used to enhance the structure. Using lines on larger numbers of categories leads to clutter.

- **Users should be able to use views without further explanation.**
  Titles, descriptions and overlays make visualizations meaningful. Use long titles that summarize the views. Use overlays for providing information about nodes. If possible, use borders to separate text and nodes.
7. Conclusion

In Section 7.1, we first present conclusions based on the research as presented in this thesis. Next, we report some findings on MagnaView in Section 7.2. Finally, we give a number of recommendations for future research in Section 7.3 and some possible improvements for MagnaView in Section 7.4.

7.1. How to Visualize Business Information Using Treemaps

We have shown that business information can effectively be transformed into tree structures that are suitable for representation using visualizations based on treemaps. The concept of levels proved to be useful: When generating tree structures from tabular data using levels, user can intuitively interact with and reason about the resulting tree structures.

We have generalized traditional treemaps, strip treemaps and squarified treemaps into a single algorithm. Parameters of the generalized algorithm, such as the layout algorithm, the directions of the strips and the sort method, can be set per level. This way, layouts that combine the strong points of the base treemap algorithms can be generated.

Next, we have defined four properties that characterize treemaps. Strict space-filling (SSF) indicates that treemaps use all available area for displaying leaf nodes. Inside parent (IP) and non-overlapping (NO) guarantee a clear mapping of tree structure to area. Uniform density (UD) ensures that quantitative comparison between nodes is possible.

Next, we have proposed extensions to the generalized treemap algorithm. Allowing size functions other than sum of children, most notably constant size, can lead to violation of UD. To counter this we have relaxed SSF to space-filling (SF), which holds if all nodes, except the root node, use all available space. Thus not only leaf nodes but also inner nodes can use space. Auto-size borders adjust the density of the child nodes such that UD holds. Often, these borders enhance the structure of the visualization, and thus, the area used by the borders is not wasted.

Another extension is the use of transformations in pixel space. In particular, we have presented mirror, pyramid and polar transformations. Polar transformations enable the use of radial layouts, while only needing to create the layout algorithms for Cartesian layouts. This is an effective way of offering different looking visualizations, like pie charts, without creating new layout algorithms.

These techniques can be used to create treemaps with the look and feel of business graphics, while still maintaining the advantages of treemaps. The familiar structures of business graphics, tried and tested methods for quantitative comparison of data, can be combined with the possibility of displaying both detail and overview of treemaps.
Node-link diagrams are typically not considered to be business graphics, but nevertheless are a classic example of easy to understand visualizations. We have shown how node-link diagrams of tree structures can be created by drawing lines on top of treemaps.

The combination of these features unifies many visualizations into a single framework. Extended treemaps can be used to produce Minard’s Tableau, bar and column charts, pie charts, stacked columns, 100% stacked columns, parallel columns, node-link diagrams, dendrograms, radial node-link diagrams, radial dendrograms, mosaic displays, icicle plots, sunburst, interring, matrices and of course traditional treemaps, strip treemaps and squarified treemaps.

Besides these visualizations, many new variations and combinations are possible, which proves useful in practice because there is no single best visualization that fits every case. Furthermore, combinations arise that are structured on a high-level and still maintain quantitative comparison of many records. Finally, we have examined the spectrum of visualizations in a structured way and derived some guidelines for creating effective visualizations.

### 7.2. MagnaView

All proposed extensions are implemented in a system called MagnaView which runs on common hardware and Windows operating systems. MagnaView can be started as a stand-alone application or as part of an information system. In the latter case users can initiate actions in the information system using MagnaView. MagnaView can read generic tabular data from various data sources, like txt files, xml files, Excel files, Access files and ODBC connections. After loading the data, MagnaView can quickly generate tree structures which are visualized using the generalized and extended treemap algorithm as proposed in this thesis. Users can modify all setting using the GUI and use MagnaView for explorative data analysis. Settings can be saved into project files, which contain specifications of how to load and visualize the data.

The implementation has a small footprint. The extensions do not add much source code, while still leading to many new possibilities. Only a small set of primitives is needed for displaying interactive visualizations. Porting this set to other platforms is sufficient for displaying and interacting with the complete range of visualizations. This has been successfully carried out for a web version.

MagnaView is used commercially and we have received much positive user feedback. Users quickly understand the benefits that treemaps add above ordinary business graphics. One user, a board member of an international law firm, summed it up nicely: “MagnaView really X-rays your organization!”
7.3. **Recommendations for Future Research**

This section lists recommendations for future research that is related to the research presented in this thesis.

- **Reconsider inside parent (IP) and non-overlapping (NO) properties**
  Letting go of the strictly space-filling property of treemaps resulted in new opportunities and insights. Reconsidering the IP and NO properties may also lead to useful results. Especially dropping NO seems interesting, it may lead to visualizations that use the same area multiple times, and thus are more efficient than space-filling visualizations, while, hopefully, still remaining effective.

- **Extend to other visualizations**
  We have created treemaps that have the look and feel of bar charts, pie charts and node-link diagrams. Future work could look into extensions that allow the creation of other familiar visualizations like line graphs and scatter plots or less familiar visualizations like beam trees and parallel coordinates. This leads to many new combinations, like, for instance, icicle plots that contain line graphs of time-series, or parallel coordinates that use squarified treemaps to layout each axis.

- **Incorporate other techniques**
  Several other techniques can be used to further enhance visualizations, most notably:
    1. Animation, which makes transitions between two visual states more clear,
    2. Distortion, which is used to provide one or more focus areas while still showing the context in a single visualization,
    3. Interaction, which is used to present information to users upon request.
  Future work could look into generalizations of these techniques such that they are applicable to all visualizations that can be generated using our framework.

- **Transformations**
  Future work could look into new transformations like rotations, spirals and space-filling curves. Furthermore, the current implementation could be improved on by better allocation of temporary bitmaps; instead of always allocating a temporary bitmap which has the same size as the non-transformed bitmap, it is better to allocate a temporary bitmap which has the dimensions of the transformed area.

- **Layout algorithms**
  Besides the traditional treemap, the strip treemap and squarified treemap layout, we have added matrix layouts. Future work could look more into different layout algorithms, for instance,
    1. The treemap layout algorithms can also be used to make many kinds of node-link diagrams. Examination of other layout algorithms for node-link diagrams can lead to new insights that can also be advantageous for treemaps.
    2. Layout-algorithms that guarantee particular aspect ratios would be useful, for instance for grouping pie charts. In general this conflicts with space filling.
layouts, but weakening this requirement and aiming for high fill ratios could lead to algorithms that are nearly space-filling but still guarantee precise aspect ratios.

3. Symmetric layouts have desirable perceptual properties. Preliminary experiments using symmetry are hopeful and suggest that local symmetry can be obtained using simple heuristics. The global implications on complete visualizations are still unknown.

- **Minimal uniform density**
  Using constant size and constant size borders can result in differences in density between nodes. Auto-size borders can be used to adjust for this such that uniform density holds again. These auto-size borders essentially waste area; the area is not used for expressing quantity or structure; the data density of the visualization is higher than required.
  Instead of using auto-size borders, it is also possible to adjust the sizes of nodes. The size of a high density node is inflated, such that the node is allocated more area and the density becomes lower. Because more area is allocated for the inflated node, less area remains for the other nodes. This heightens the density of the other nodes, but can decrease the maximum density and thus the required uniform density.
  Minimal uniform density is the state in which it is not possible to further lower the uniform density and thus lower the waste of area. Preliminary research using an iterative approach in which a single node is enlarged in each iteration shows that gains are possible. Future work could look into the limitations caused by constraints, i.e., it is not always possible to obtain layouts that satisfy all constraints (borders). Future work could also look into more efficient algorithms to obtain minimal uniform density, for instance updating multiple nodes per iteration.

- **User study**
  MagnaView can create various visualizations, like basic treemaps, pie charts, icicle plots, node-link diagrams and variations on these, but it is unclear which visualization is most suitable. MagnaView offers an environment that is exactly the same for each visualization. This could be used for user studies which give more insight in the strong and weak points of each visualization method.

- **Automatic view generation**
  The final look and feel of a view in our framework is determined by many settings. Creating good views can be cumbersome, even for expert users. Therefore, it would be beneficial to (partially) automate the generation of views. Complete automation seems unlikely because the current interest of users is not known by the system. Therefore, selecting the levels of the tree structure that is generated will most likely remain a user task. The settings that determine the layout, on the other hand, seem good candidates for automation. Whether local heuristics for these settings will lead to useful visualizations is unknown at the moment. Automatic view generation also enables the use of per node settings as
opposed to per level settings we currently use. This could lead to visualizations that better match heterogeneous data.

7.4. **Possible Improvements on MagnaView**

This section lists some possibilities for improvements on MagnaView and research opportunities that are less related to the research presented in this thesis.

- **Improving loading of data**
  Users can only use a visualization after they have successfully loaded their data. Currently, both the user interface for and performance of loading data need improvements.

- **Supporting bigger datasets**
  There are always bigger datasets. While there are practical limits to the number of nodes that can be visualized in a single image, it still is advantageous to be able to handle larger datasets. Currently, a dataset with 200,000 records can be visualized without problems. Future work should raise this to about 2,000,000 records.

- **Color**
  Color is one of the most eye catching properties of a visualization; color can make or break a visualization. The color options we provide to users are minimal, i.e., users can select colors by hand, or choose colors for the minimum and maximum values and let MagnaView interpolate the colors for the values in between. Future work should look into more flexibility in color use, like using expressions to determine the colors of nodes. Effort should also be made to better help users selecting the right colors by offering presets and hints.

- **Labels**
  Positioning and rendering of labels seems trivial, but has some real problems in practice, which future work could look into, for instance
  1. Overlays can contain labels, but the underlying area becomes obfuscated. Labels should never overlap with important parts of a visualization, including other labels,
  2. Text is most readable in horizontal rectangles that have high aspect ratios. Using labels in vertical oriented visualizations, like icicle plots, can be problematic.
  3. Transformations do not combine well with labels, for instance, leading to labels that are positioned upside down. Therefore, labels are always rendered non-transformed, but it is not clear what location to choose in this case.
  4. Rendering of labels should result in clearly readable labels. There should be sufficient contrast between the label color and the background color. A reasonable solution to this is the use of text shadows, but this does not help in all cases.
- **Search function**
  Visualizations excel in letting users visually search for outliers and associations. However, finding specific nodes can be hard. Future work could look into search functions that enable users to quickly find nodes of interest. Related work uses text search functionality and sliders to select ranges for attributes. Free text search, like internet search engines use, is also a promising technique to incorporate.

- **Sorting**
  Users typically want to sort bar charts on descending size; matrix layouts become much clearer if the axes are sorted in a way that diagonalizes the matrix. In general, calculating the optimal order of nodes is a NP hard problem. However, the potential gains are substantial, so future work could look into heuristics for sorting nodes.

- **Visual querying**
  Real-world experiences suggest that users who have trouble using SQL to perform queries on a database have less trouble using MagnaView for equivalent queries. This is under the assumption that generic SQL queries are available for loading data into MagnaView, such that users can perform more specific queries in MagnaView. Future work could look into the possibilities of combining SQL like queries and visual queries.
References


[36] B. Shneiderman, "Designing for fun: how can we design user interfaces to be more fun?", Interactions 11, no. 5, pp. 48-50.


